THE UNIVERSITY *of* EDINBURGH

Constraint-Based Specifications for System Configuration

John A. Hewson

Doctor of Philosophy

Centre for Intelligent Systems and their Applications
School of Informatics
University of Edinburgh

2013

Abstract

Declarative, object-oriented configuration management systems are widely used, and there is a desire to extend such systems with automated analysis and decision-making. This thesis introduces a new formulation for configuration management problems based on the tools and techniques of constraint programming, which enables automated decision-making.

We present ConfSolve, an object-oriented declarative configuration language, in which logical constraints on a system can be specified. Verification, impact analysis, and the generation of valid configurations can then be performed. This is achieved via translation to the MiniZinc constraint programming language, which is in turn solved via the Gecode constraint solver. We formally define the syntax, type system, and semantics of ConfSolve, in order to provide it with a rigorous foundation. Additionally we show that our implementation outperforms previous work, which utilised an SMT solver, while adding new features such as optimisation.

We next develop an extension of the ConfSolve language, which facilitates not only one-off configuration tasks, but also subsequent re-configurations in which the previous state of the system is taken into account. In a practical setting one does not wish for a re-configuration to deviate too far from the existing state, unless the benefits are substantial. Re-configuration is of crucial importance if automated configuration systems are to gain industry adoption. We present a novel approach to incorporating state-change into ConfSolve while remaining declarative and providing acceptable performance.

Acknowledgements

Firstly, I would like to thank my supervisors, Paul Anderson and Andy Gordon, for their time, support, and wisdom. Their enthusiasm for research is clear and compelling. I would like to thank Paul Jackson for his guidance and insight, at a critical moment.

This thesis would not exist were it not for the support and love of my family. I would like to thank my parents for their unquestioning faith and my wife, Kelley, for her counsel, patience, and encouragement.

I'd like to thank those scholars with which I have had particularly insightful conversations about this thesis: Alva Couch, Tim Nelson, H. Herry, and Ed Smith. Thanks too to those researchers in industry: Sharad Singhal and Andy Farrell from HP Labs, Alois Haselboeck from Siemans, Austin Donnelly from Microsoft Research, and John Wilkes from Google. A special thank you to Sharad for providing the Cauldron binaries for benchmarking.

This work was funded by Microsoft Research through their European PhD Scholarship Programme.

Declaration

I declare that this thesis was composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or professional qualification except as specified.

(*John A. Hewson*)

Related Publications

Some of the contents of this thesis are based on the following papers:

- J. A. Hewson and P. Anderson and A. D. Gordon. Constraint-Based Autonomic Reconfiguration. In *Proceedings of the Seventh IEEE International Conference on Self-Adaptive and Self-Organizing Systems (SASO 2013)*, 2013.

- J. A. Hewson and P. Anderson and A. D. Gordon. A Declarative Approach to Automated Configuration. In *Proceedings of the 26th Large Installation System Administration Conference (LISA'12)*, 2012.

- J. A. Hewson and P. Anderson. Modelling System Administration Problems with CSPs. In *Proceedings of the 10th International Workshop on Constraint Modelling and Reformulation (ModRef'11)*, pages 73–82, 2011.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

The role of the system administrator has changed significantly over the past twenty years. The most visible difference being the rise of declarative configuration languages and tools. Declarative configuration allows one to describe systems in terms of a desired goal state, on a per-machine basis. Centralised servers compose configurations for individual machines from declarative classes or "aspects", acting as both a point of control and as a catalogue of network truth. Modern systems have progressed far from being managed with ad-hoc shell scripts and a human being at a computer terminal.

The driving force behind this change has predominantly been scale. Computer systems are larger than ever, with cloud computing and infrastructure-as-a-service datacenters bringing new challenges and higher administrator to machine ratios. Centralised, declarative tools provide other advantages though. Industry practice has shifted towards repeatable, highly automated, resilient tools and processes, as embodied by the DevOps movement.

Current declarative configuration tools are still organised around the concept of a machine, a single "node". As more software becomes network-based and virtualised the individual machine is shifting from being the end point of a configuration to the starting point. The question of how to manage the relationships between machines arises. On the network relationships are dynamic; a "static" configuration, a single point in time, is neither resilient nor flexible. A machine failure requires an administrator to manually alter the central configuration.

A "dynamic" configuration must express more than just a single configuration and allow logical entities beyond single machines to be described. It is the invariant constraints on the relationships between services, virtual machines, networks, and other high-level components which are of interest. Object-orientation has been used

successfully in configuration management and is used to various degrees in current declarative tools. Some efforts have been made to combine logic programming with object-orientation for the purpose of validating configurations and generating valid configurations from a more loosely specified policy.

This desire to state a problem as a set of logical constraints and have the computer automatically find a solution is reflected in another area of computer science: constraint programming. Constraint programming allows one to describe a problem in terms of variables with finite domains and constraints over those variables. Constraints may be drawn from logic, arithmetic, or set theory. The search strategy need not be described: it is the task of the constraint solver to provide a good search method for each constraint it permits. Indeed, we can see that constraint programming is a declarative paradigm.

In practice, constraint programming alone is insufficient to meet the needs of system administrators. It takes an expert user and a significant refactoring of the problem from a natural description to one which is practically solvable. The purpose of this thesis is to close the gap between logical languages and configuration languages. We find inspiration in previous work by HP Labs, devising a performant formalised alternative based on constraint programming, with a natural extension to constraint optimisation.

Existing configuration approaches treat configuration problems as one-off tasks: initial configurations of a new system, starting from scratch. In practice the majority of configuration tasks are incremental, starting from some existing state and applying changes which take the existing configuration into account.

Take for example, the common problem of assigning virtual machines to physical hosts. After a system has been configured initially, it is desirable for subsequent reconfigurations to take into account the current allocation of virtual machines so as not to move virtual machines unnecessarily from one physical machine to the next, as illustrated in Figure 1.1. When using a constraint-based approach such moves are to be expected. The constraint solver will simply follow the quickest path to satisfaction available to it.

Thus it is necessary to inform the solver about the previous state of the system and how it affects subsequent configuration decisions. In this thesis we devise a novel method to capture state and expose it within a declarative configuration language.

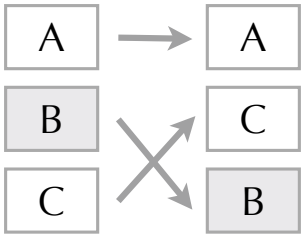

Figure 1.1: A reconfiguration of the allocation of virtual machines A–C to three physical hosts. Virtual machines B and C have switched hosts unnecessarily as an artefact of constraint-based reconfiguration.

1.1 Research Questions

This thesis originates from a research proposal by Paul Anderson and Michael Fourman accepted by Microsoft Research for a PhD scholarship entitled *constraint-based specifications for system configuration*. The research objectives are as follows:

1. Identify where constraint-based approaches to system configuration may be advantageous.
2. Investigate the suitability of existing constraint-based techniques for solving configuration problems.
3. Propose a method which can apply the identified constraint-based techniques to the identified configuration problems.
4. Design and implement a demonstration tool based on the proposed methods.
5. Evaluate the tool against the identified configuration problems.

In particular, existing configuration tools and research prototypes have lead to several open questions, which there is a need to address:

6. Soft constraints, which need not be fully satisfied, need to be able to be modelled and solved.
7. Small changes to the configuration problem should result in small changes to the resulting solution.
8. The configuration interface needs to be appropriate for the intended users. Thus novel features need to introduce a minimal amount of complexity.

1.2 Contributions

The main contributions of this thesis are divided into two parts. Firstly, we answer the research questions which concern one-off automated configuration tasks similar to those seen in current research. Secondly, we address the open questions surrounding reconfiguration.

1.2.1 Configuration

With regards to configuration, the main contribution is to formally define and evaluate a configuration language based on constraint satisfaction, specifically:

1. Define ConfSolve, a constraint-based object-oriented configuration language.
2. Define the translation of the language to a constraint satisfaction problem.
3. Demonstrate that the language can be used to model problems identified in previous work based on boolean satisfiability.
4. Show that translated models can scale to problems of a useful size.

1.2.2 Reconfiguration

With regards to reconfiguration, the main contribution is to extend our constraint-based language with novel state-aware primitives based on constraint optimisation, specifically:

1. Define an extension to the ConfSolve language which incorporates reconfiguration via state-aware constraints.
2. Define the translation of our extensions to a constraint satisfaction problem.
3. Show that ConfSolve reconfiguration primitives can offer a performance benefit over a from-scratch configuration and a naive reconfiguration heuristic.
4. Show that translated models can scale to reconfiguration problems of a useful size.

1.3 Thesis structure

The remainder of this thesis is structured as follows:

Chapter 2: ***Background*** Provides background material covering configuration management and constraint satisfaction. State-of-the-art model-driven approaches to configuration management are summarised and discussed. Constraint satisfaction tools and techniques used in the thesis are presented in detail.

Chapter 3: ***Related Work*** Discusses existing work which is relevant to this thesis. It is divided into three major sections: automated configuration, automated reconfiguration, and constraint modelling languages. Each of the first two sections are divided according to the constraint-solving approach used in the work.

Chapter 4: ***ConfSolve by Example*** Provides an overview of the ConfSolve language developed in this thesis. This informal description is intended to demonstrate each of the language features in a practical manner.

Chapter 5: ***The Syntax and Semantics of ConfSolve*** We define the abstract grammar of the ConfSolve language and its output format and define its translation to a constraint satisfaction problem encoded in MiniZinc.

Chapter 6: ***Evaluation of Configuration with ConfSolve*** The performance of ConfSolve is measured using several system configuration problems. A large virtual machine allocation problem and several smaller problems from previous work, which we benchmark against.

Chapter 7: ***Extending ConfSolve for Reconfiguration*** An extension to the basic ConfSolve language which permits reconfiguration described. We present a conceptual overview, example, syntax extensions, and a description of the translation of the extended language to MiniZinc.

Chapter 8: ***Evaluation of Reconfiguration with ConfSolve*** The reconfiguration extensions for ConfSolve are evaluated against several system configuration problems. The problems from Chapter 6 are expanded to cover reconfiguration and the three scenarios of parameter change, migration, and a combination of the two.

Chapter 9: ***Conclusions*** Draws conclusions and outlines various directions for future work.

Chapter 2

Background

This chapter provides background material covering configuration management and constraint satisfaction. Popular model-driven approaches to configuration management are summarised and discussed to give an insight into the state-of-the-art. Constraint satisfaction is presented in detail, with the tools and techniques used in this thesis explained and differentiated from similar techniques.

2.1 Configuration management

There are a wide range of tools which facilitate the task of system configuration. System administrators have developed and adopted competing approaches which reflect the changes seen in programming languages over the same period of time. That is, a shift from basic scripting languages to object-oriented modeling and finally a more lightweight declarative approach. This centralised, abstracted, and automated approach to system configuration has become known as *configuration management*.

2.1.1 Common Information Model (CIM)

Object-oriented programming techniques have influenced configuration management. The Common Information Model (CIM) created in 1999, provides an object-oriented meta-model for describing and managing computer systems in a standardised manner [Distributed Management Task Force, 2010a]. It is the basis for a number of other standards which define protocols for interacting with CIM objects on a machine in an open, cross-platform manner. Remote management of CIM-described resources is possible using standardised web service protocols, with Web-Based Enterprise Manage-

```
[abstract]
class Win32_LogicalDisk
{
   [read]
      string DriveLetter;

   [read, Units("KiloBytes")]
      sint32 RawCapacity = 0;

   [write]
      string VolumeLabel;

   [Dangerous]
     boolean Format([in] boolean FastFormat);
};
```

Figure 2.1: A MOF file, describing a Windows logical disk. Types, read and write properties, and methods are present. [Distributed Management Task Force, 1999]

ment [Distributed Management Task Force, 2010c] and WS-Management [Distributed Management Task Force, 2010b] being widely used. However, CIM does not provide a centralised configuration system; it is a standard description-method and protocol rather than a configuration system.

A key component of CIM is the CIM Schema, which defines extensible, standardised descriptions for an extensive range of hardware and software components. However, CIM models are primarily used for proprietary vendor-specific enterprise systems which extend standard CIM classes. With so many non-standard CIM classes the value of standardisation is not clear.

CIM includes a language to describe objects, the Managed Object Format which is based on the Interface Definition Language (IDL) for language-independent interface specification, typically used for remote procedure call. MOF is capable of describing meta-models and schema, as well as classes, properties and method signatures. A sample MOF file is shown in Figure 2.1.

CIM adoption has not been as widespread as perhaps it could have been. The complexity of the protocols and schema, and the lack of need for such a system outside of an enterprise environment has led to a situation where only large enterprise software from companies such as IBM and HP provide CIM interfaces, and Microsoft Windows, which has included an enterprise-targeted CIM interface since 2000 [Microsoft

Corporation, 2010].

There is a notable absence of CIM support in open-source software such as Linux. The complexity of implementing CIM presents a significant opportunity cost to developers who would otherwise be contributing their time to developing the core functionality of their software [Brasher and Schopmeyer, 2006]. Furthermore, the CIM object model allows the creation of object instances and the execution of instance methods in an imperative manner and allows the existence CIM objects which do not provide a declarative interface. This results in an imperative rather than declarative configuration process.

2.1.2 Declarative configuration

The past twenty years has seen the rise of *declarative* configuration, in which the goal state of the system is specified, rather than the steps required to achieve it. This concern with the "what" rather than the "how" sets declarative configuration apart from scripts, conditional actions, and calling methods on objects. Indeed, we can see that these traditional forms of configuration are *imperative*. All declarative configuration systems are based on two important principles:

Idempotence requires that when a configuration action is successfully repeated, the outcome will be identical. This is important when using low-level abstractions, such as a command which adds a line to a configuration file. A command which ensures that a given line exists must instead be used, otherwise the file may contain the same line numerous times. Idempotence is a desirable property because recovery from a failure becomes possible simply by executing the configuration commands again; presuming that the commands themselves do not cause the error. In practice, idempotence acts as a substitute for *atomicity*, which is not achievable for configuration actions that cannot be rolled-back [Kanies, 2003].

Convergence is a property of a system which moves eventually to a goal state. The system does not have to be in a consistent, configured state at any point prior to convergence. In such a system it is the continuous re-application of idempotent configuration commands which enable the system to repeatedly reconfigure until the goal state is reached. The are two advantages to this approach. Firstly, it is possible for the system to recover from the failure of a given configuration by simply re-running the configuration commands. For example, if a file on an unavailable remote server is required,

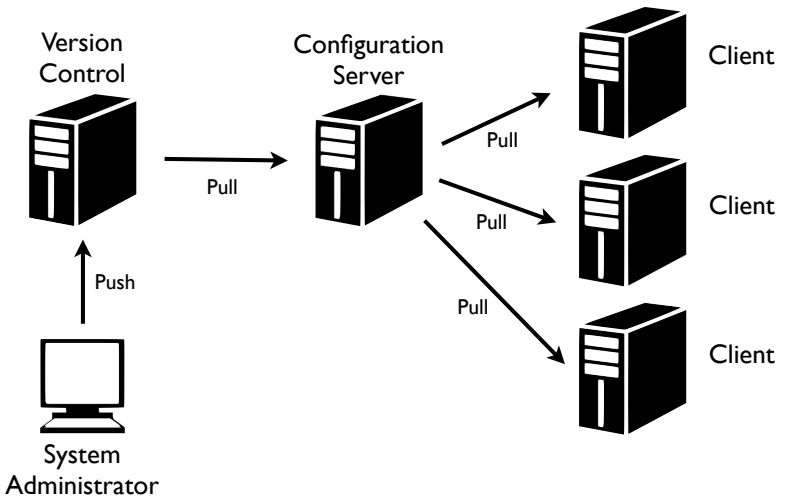

Figure 2.2: A typical declarative configuration system. The system administrator authors a declarative specification which is stored in version control. The configuration server periodically retrieves the latest revision and computes the configuration for each of its clients. Clients periodically retrieve their configuration from the server in a voluntary manner.

then when it is eventually available the configuration process will continue. Secondly, the concept of convergence applies equally well to both individual machines and distributed systems. For example a server may require configuration to allow a client to access it and the client may require configuring with the address of the server. As one is inevitably configured before the other, there is an intermediate undefined state, during which the system is converging towards the goal state of both client and server being configured. In a convergent system we are not interested in the order in which idempotent commands are executed, as long as they eventually converge.

2.1.3 Declarative tools and languages

There are several declarative configuration tools in use today. These tools are conceptually similar and share the common goals of providing abstraction, encapsulation, and site-wide automation. Each has a centralised architecture, where a server is responsible for distributing the appropriate configuration files to a specific machine. The primitives provided by each configuration languages are comparatively low-level. The concept of a logical component: a “bundle” of files or operations, exists to some extent in each system. Lower-level primitives are combined with a mechanism for dividing the machines to be configured into different “classes”, based on some machine-specific variable, such as which operating system is installed. This allows the administrator to

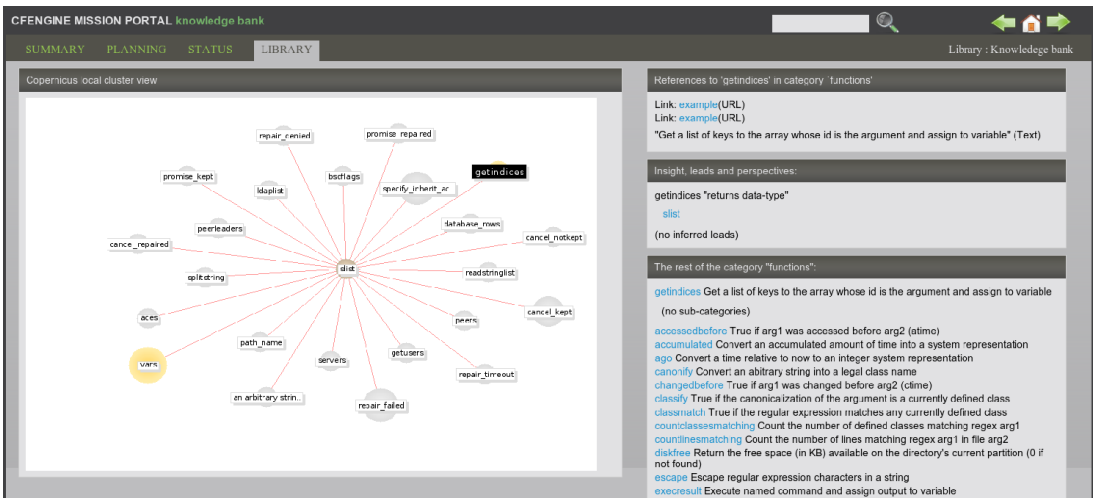

Figure 2.3: CFEngine management interface, showing a knowledge management view.

assign specific roles to various machines, or groups of machines. The configuration system then combines the configuration aspects for the various classes into which each machine falls, and so dynamically produces machine-specific configurations. Figure 2.2 shows the architecture typical of such systems.

The remainder of this section will examine several popular declarative configuration tools. Each system has its own strengths and weaknesses, its own terminology, its own abstractions over underlying resources, its own protocols, and its own configuration language.

CFEngine

CFEngine was initially developed in 1993 and was one of the first declarative configuration systems [Burgess et al., 1995] along with LCFG [Anderson and Scobie, 2000]. It follows the centralised model from Figure 2.2, and provides a declarative configuration language as well as a web-based management interface, shown in Figure 2.3 which facilitates knowledge management and auditing.

CFEngine is grounded in a theoretical model known as *promise theory*, which was developed to describe the original CFEngine's use of idempotent and convergent operations. In CFEngine new configurations are pulled from the server by the client, rather than pushed, which results in a voluntary process. Promise theory abstracts both constraints over a local machine and between remote machines, enabling distributed configuration processes to be modelled as promises to perform certain configuration operations. While promise theory offers insights into distributed and delegated config-

```
bundle agent set_root_password
{
files:
"/etc/passwd"
  comment => "Set the root password",
  edit_line => set_user_field("root",2,"xyajd673j.ajhfu");
}
```

Figure 2.4: A CFEngine bundle, showing a bundle with a single `file` promise which sets the `root` user's password [CFEngine AS, 2008].

```
bundle agent mysql_running {
  commands:
    "/etc/init.d/mysql start"
}
```

Figure 2.5: CFEngine bundle with an imperative command to start MySQL. This is not the recommended solution to this problem, but there are no measures to prevent the user from writing such a configuration.

uration processes, it does not contain any formalisms which are of use in this thesis.

A CFEngine configuration files consists of a series of promises, grouped into bundles. A bundle is a collection of promises which are applied together, and enforced either on the client or the server. The body of a bundle is a template macro which consists of a series of promises divided by type; Figure 2.4 shows a simple bundle which sets the password of the `root` user. CFEngine's low-level primitives such as file editing and shell command execution can lead to the creation of non-convergent operations. For example, in Figure 2.5 a promise which executes a shell command is shown.

Puppet

Puppet is a popular configuration tool created in 2005 to supersede CFEngine [Puppet Labs, 2008]. It follows the same centralised architecture and conforms closely to the original CFEngine's resource-based model, rather than modeling in terms of promises. Like CFEngine it features a sophisticated web-based management interface, shown in

Figure 2.7. As we have already seen a basic example of CFEngine configuration, we provide an extended concrete example of Puppet configuration in Figure 2.6.

2.2 Constraint satisfaction

This section introduces the concept of constraint programming and gives an overview of theoretical and practical aspects before, then presents the constraint solving and modelling tools used in this thesis.

2.2.1 Constraint programming

Constraint programming is an approach to solving combinatorial problems in which relations between variables are expressed as constraints. It is a declarative paradigm; constraints are logical invariants rather than individual steps to be executed. Constraint programming emerged from *constraint logic programming* in the late 1980s, in which constraint handling is embedded in a host language. Extended versions of Prolog were among the first implementations [Rossi, van Beek, and Walsh, 2006].

Constraint satisfaction problem

The central theoretical concept of constraint programming is that of the *constraint satisfaction problem* (CSP). A CSP consists of a finite set of variables, each associated with a finite domain from which its values may be drawn, and a set of constraints that restrict the possible values that variables can take simultaneously. A solution is an assignment of values to variables which satisfies all constraints. In the general case CSPs are NP-complete, as they are a generalisation of boolean satisfiability (SAT), the first known NP-complete problem.

Constraint optimisation problem

An common extension of the CSP is the *constraint optimisation problem* (COP), in which a CSP is augmented with a cost function which must be minimised or maximised. Equivalent formulations include MAX-CSP and Weighted CSP. The constraint optimisation problem is NP-hard, since it involves solving CSP, which is NP-complete, as a sub-step. Likewise, its boolean counterpart, MAX-SAT is also NP-hard.

```
# ntp.pp

class ntp {
  case $operatingsystem {
    centos, redhat: {
      $service_name = 'ntpd'
      $conf_file    = 'ntp.conf.el'
    }
    debian, ubuntu: {
      $service_name = 'ntp'
      $conf_file    = 'ntp.conf.debian'
    }
  }

  package { 'ntp':
    ensure => installed,
  }

  service { 'ntp':
    name      => $service_name,
    ensure    => running,
    enable    => true,
    subscribe => File['ntp.conf'],
  }

  file { 'ntp.conf':
    path    => '/etc/ntp.conf',
    ensure  => file,
    require => Package['ntp'],
    source  => "puppet:///modules/ntp/${conf_file}",
  }
}
```

```
# init.pp

node server1 {
  include ntp
}
```

Figure 2.6: A complete Puppet class for configuring NTP [Puppet Labs, 2011]. The configuration varies between operating systems, one for CentOS and Red Hat, another for Debian and Ubuntu. This configuration specifies that the NTP binary package must be installed, that the NTP service should be running and configured to use the `ntp.conf` file downloaded from the Puppet server. A single machine, `server1` is configured to use the NTP class.

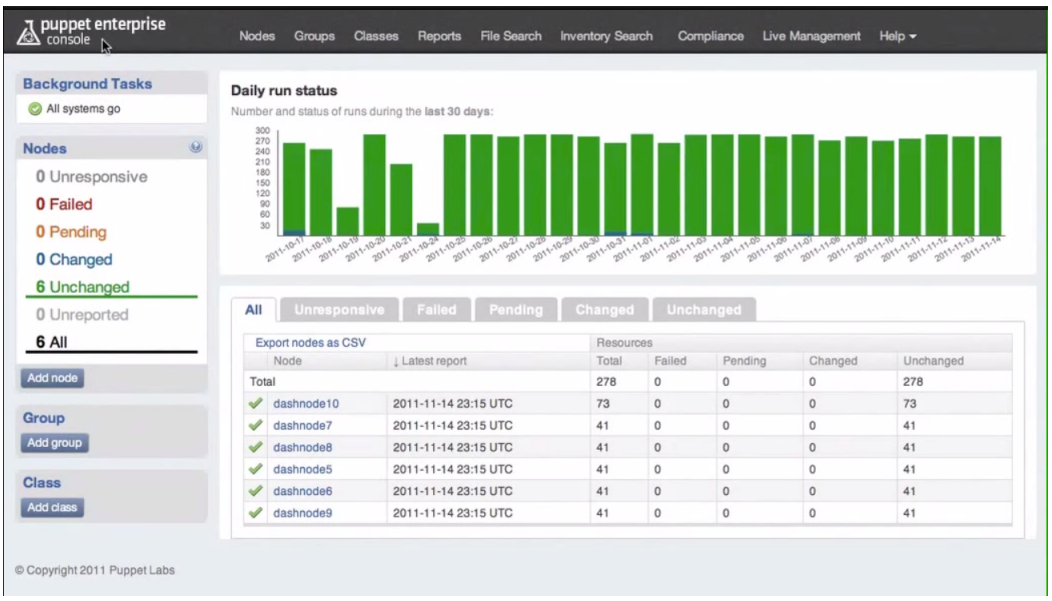

Figure 2.7: The Puppet management interface, showing successful runs across an entire site.

Completeness

When we talk of *completeness* in the context of constraint programming, it is in reference to the search procedure being used. A complete search procedure is an exhaustive one, which covers the entire search space. This does not mean that every solution has been individually tested though, as inference may be used to prune the search space. An example of an incomplete search method is stochastic local search, in which the search space is explored randomly for a fixed period of time. Thus even for a decidable logic, the solver may return the result "unknown".

Search completeness should not be confused with *logical completeness*. Consider a search procedure for a (hypothetical) non-finite-domain CSP with constraints in first-order logic. It will always have an incomplete search procedure because of the undecidability of first-order logic, irrespective of the fact that there exist logically complete deduction systems for first-order logic.

2.2.2 Constraint solving: Gecode

Constraint satisfaction problems are solved via a *constraint solver*. In this thesis we make use of the Gecode solver, an efficient, award-winning open-source constraint solver written in C++ [Gecode Team, 2006]. Gecode solves problems with finite domains, using an exhaustive (*i.e.*, complete) search.

Backtracking

The search procedure is built upon the process of *backtracking*. This is a recursive algorithm where each variable is assigned a value in turn. Upon each assignment the constraints on that variable are checked for consistency. If the assignment is consistent, then a recursive call is made to assign the next variable. If the assignment is inconsistent then the next value from the variables domain is tried. If their are no remaining values, then the algorithm backtracks.

Constraint propagation

Backtracking is inefficient, and though modern constraint solvers will backtrack when they need to, it is *constraint propagation* which is a form of inference, and provides most of the solving ability. After a successful variable assignment the domains of each variable that shares a constraint with the current variable are checked, and any values incompatible with the newly assigned value are removed. This continues recursively until no domains need pruning. If a domain is empty then backtracking is triggered. Many CSPs can be solved entirely via propagation.

Modern CSP solvers rely upon the use of various *propagators* to perform constraint propagation. Each class of constraints has its own propagator that implements an efficient inference algorithm for domain pruning, such as a propagator for linear arithmetic. This gives CSP a potential advantage over SAT similar to that of SMT; the solver is equipped with higher-level information about a problem and can employ more efficient search strategies.

Branch and bound

So far we have discussed approaches to constraint satisfaction. In order to instead solve a constraint optimisation problem, a *branch and bound* algorithm is employed along with the methods above. This stores the cost of the best solution found during the search and compares it to candidate partial solutions as they are encountered. When a partial solution is reached which cannot be completed to achieve a better cost than the stored best, then backtracking is triggered. This avoids an unnecessary search into fruitless branches of the search tree, and is the method adopted by Gecode.

Set domains

Variables may be integers, arrays, or sets of integers. Yet the domain of a set variable is the cartesian product of its element type's members, which is too large to store in memory, especially as backtracking keeps copies of domains. This problem is solved by requiring sets of variable cardinality to have a finite element type. A set of 1..5 is allowed, but a set of `int` is not, which ensures that set variables have a known upper-bound on their cardinality. Gecode and other solvers then approximate set domains as an interval, consisting of only the upper and lower bounds. These upper and lower bounds are themselves sets, as the domain stored in full would be a set of sets. Such sets are ordered based on inclusion ($\subseteq$), starting with the empty set and ending with the set of all permissible elements. For a complete description of the process, see [Gervet, 1997].

2.2.3 Constraint modelling: MiniZinc

Constraint programming consists of writing models of a problem using constraints. The constraints themselves are typically low-level, directly exposing the functionality of the solver. For example, the constraint `int_max(a,b,c)` ensures that $max(a,b) = c$ holds, where a, b, and c are integers. At this level constraints are *flat*, they do not have any nested expressions.

In order to ease the process of writing such programs, solvers typically provide a *modelling language* which provides higher-level constructs. In recent years there has been a move towards adopting a standard modelling language for constraint programming and the language *MiniZinc* has emerged as a prominent standard supported by numerous solvers [Nethercote, Stuckey, Becket, Brand, Duck, and Tack, 2007].

In MiniZinc the high-level and low-level representations of the model are separated. The MiniZinc compiler accepts a MiniZinc model as its input, and generates a low-level solver input file expressed in *FlatZinc*. The benefit of this approach is that it is relatively simple for solver authors to implement FlatZinc support, as it is close to the internal representation of constraints used by most solvers.

Take for example, the constraint $a > 5 \vee b$ where a is an integer and b a boolean. It may be modelled as follows in MiniZinc:

```
var int: a;
var bool: b;
constraint a > 5 \/ b;
solve satisfy;
```

Compiling with the MiniZinc compiler results in the following FlatZinc model:

```
var bool: BOOL____00001;
var int: a;
var bool: b;
constraint array_bool_or([BOOL____00001, b], true);
constraint int_le_reif(6, a, BOOL____00001);
solve satisfy;
```

The FlatZinc model represents a *flattened* version of the original model, in which there are no nested expressions. The term `a > 5` has been substituted with a new variable `BOOL____00001`.

Furthermore, the MiniZinc compiler has performed a number of transformations to produce a more efficient model. The logical disjunction has been replaced with the constraint `array_bool_or`, which takes an array as its first argument and ensures that at least one element has the truth value corresponding to the second argument, in this case `true`.

The sub-expression $a > 5$ has been transformed into the constraint `int_le_reif`, a *reified* constraint, in which the truth value of a constraint is reflected into a boolean variable. In this case the inequality is reversed, and the translation results in the constraint $(6 \leq \texttt{a}) \iff \texttt{BOOL\_\_\_\_00001}$.

Quantifiers

MiniZinc supports quantifiers, as well as aggregates such as `sum` and `count`. These can be applied over set literals or constants, but not set variables. This is an important limitation, as it excludes *decision variables* from being quantified.

For example, consider a set s with domain 1..3 and an integer n, it is not possible to directly write the constraint $\exists_{x \in s}\, x > n$ using MiniZinc. Instead the quantified variable must be replaced with a constant, namely the domain of s, which is 1..3. A model for

this is as follows:

```
var set of 1..3: s;
var int: n;

constraint
  exists (x in 1..3) (
    x in s /\ x > n
  );

solve satisfy;
```

This model is un-rolled by the MiniZinc compiler into a much larger FlatZinc model with three `set_in` ($\in$) and `int_le_reif` ($\leq$) constraints. It is this need to un-roll a finite number of constraints which prevents decision variables from being quantified.

Limitations

As we have seen, MiniZinc's quantifiers are quite limited in their use. Although MiniZinc offers a much higher level of modelling than FlatZinc, it is still somewhat restrictive as a language. Arrays of variables are supported, and directly correspond to low-level array primitives in FlatZinc, however they must be of a fixed size. Variables with set-domains are supported, so long as the element type of the set is finite. Such sets have their cardinality determined by the solver, and this is a powerful modelling feature. However, while arrays of set variables are permitted, the opposite is not. No other nesting of arrays or sets is possible. Sets of sets, or sets of arrays cannot be modelled directly.

We make use of MiniZinc as a convenient front-end to a constraint solver, and as a target for formalisation of an executable semantics which is not far removed from first-order logic. It's unsuitability as a modelling language for system configuration should, by this point, be clear.

Chapter 3

Related Work

This chapter discusses existing work which is relevant to this thesis. It is divided into three major sections: automated configuration, automated re-configuration, and constraint modelling languages. Each of the first two sections are divided according to the constraint-solving approach used in the work, and within each subsection work is presented approximately chronologically.

3.1 Automated configuration

This section discusses existing work related to automated configuration. It is structured based on the underlying constraint-solving technology used in the work, starting with event-driven methods and moving on to logic programming, boolean satisfiability, and constraint satisfaction.

3.1.1 Event-driven approaches

The rules for configuring a system are expressed in a policy language. Such languages can be grouped into a hierarchy [White et al., 2004]. At the bottom are *action* policies in the form "*if* (condition) *then* (action)". At the next level are *goal* policies, which allow autonomic elements to decide how to implement them. At the top level are *utility function* policies which are able to score system states based on desirability, for use in an autonomic decision process. From the point of view of system configuration, action policies are imperative, goal and utility function policies are declarative, regardless of whether or not they incorporate otherwise declarative languages, such as first-order logic.

One advantage of declarative approaches is compositionality: given requirements A and B, a solution to $A \wedge B$ is guaranteed to satisfy both A and B. However, an imperative approach does not; perhaps A must be enforced before B [Narain et al., 2008]. Furthermore, it has been observed [Couch and Gilfix, 1999] that a declarative language is necessary for a system to be convergent. An imperative language would require code to handle every possible system environment and failure scenario; existing components would have to be changed or extended each time a new rule is added [Narain, 2005].

Ponder

Ponder is an object-oriented language for specifying security and management policies for distributed systems [Damianou, Dulay, Lupu, and Sloman, 2001]. It is an event-driven, action policy language. Ponder builds on four core policy types: authorisation policies, event-triggered obligations, refrain policies, and delegation policies. Constraints act to limit when policies are applicable, while meta-policies define which policies are themselves permitted. Ponder is also capable of capturing roles, relationships, and management structures pertaining to policy users. It has been successfully combined with CIM and used to configure Linux routers [Lymberopoulos, Lupu, and Sloman, 2004].

Object Constraint Language (OCL)

The Object Constraint Language (OCL) is the constraint language of UML, and is an Object Management Group standard [Object Management Group, 2006]. It was standardised without a precise formal semantics, which were later provided [Richters and Gogolla, 1998], though weaknesses in the language were not able to be rectified. OCL is built on the concepts of class invariants and method pre- and post-conditions and is thus an action policy language. OCL expressions may incorporate the full power of first-order logic and are necessarily undecidable.

Nevertheless, OCL is amenable to automatic analysis in restricted forms. Analysis typically consists of the verification of class invariants against method pre- and post-conditions. UM2Alloy can translate an OCL model with bounded domains into Alloy for verification, though it does not support postconditions [Anastasakis, Bordbar, Georg, and Ray, 2007]. UMLtoCSP translates a bounded OCL model into a CSP which is solved via ECLiPSe [Cabot, Clarisó, and Riera, 2008]. Strictly speaking this is a constraint logic program (CLP) and not a pure constraint satisfaction prob-

lem (CSP). A translation of OCL to first-order logic allowed for subsequent checking of the formula with a SAT solver, and was able to automatically verify unbounded models [Clavel, Egea, and García de Dios, 2010].

Service Modeling Language (SML)

The Service Modeling Language (SML) was a Microsoft-driven attempt at producing a standard metamodel and language for capturing complex configurations, including constraints, which resulted in a W3C standard [W3C Members, 2007]. SML relies heavily on XML, utilising elements, attributes, namespaces, XML schema, XPath, XSLT transformations, and using Schematron for constraints. This is somewhat at odds with the contemporary view that next-generation configuration languages should be simple in order to avoid mistakes [Anderson, 2008]. SML was quickly abandoned by Microsoft, citing performance and scalability issues [Vambenepe, 2008]. An example of SML is given in Figure 3.1.

```
<xs:complexType name="IPAddress">
  <xs:annotation>
    <xs:appinfo>
      <sch:schema xmlns:sch="http://purl.oclc.org/dsdl/schematron">
        <sch:ns prefix="tns" uri="urn:IPAddress" />
        <sch:pattern id="Length">
          <sch:rule context=".">
            <sch:assert test="tns:version != 'V4' or count(tns:address) = 4">
              A v4 IP address must have 4 bytes.
            </sch:assert>
          </sch:rule>
        </sch:pattern>
      </sch:schema>
    </xs:appinfo>
  </xs:annotation>
  <xs:sequence>
    <xs:element name="address" type="xs:byte" minOccurs="4" maxOccurs="4" />
  </xs:sequence>
</xs:complexType>
```

```
<myIPAddress xmlns="urn:IPAddress">
  <address>192</address>
  <address>168</address>
  <address>100</address>
  <address>1</address>
</myIPAddress>
```

Figure 3.1: An SML schema describing an IP address (above) and its usage in a model (below) [W3C Members, 2007].

3.1.2 Logic programming

There have been numerous efforts to translate policy languages into logical formulae, from which induction may be performed to generate valid configurations. These predominantly make use of the logic programming language Prolog, which is based on Horn clauses, a subset of first-order logic.

Couch & Gilfix were early advocates of the use of Prolog as a platform-independent language for specifying goal policies, which could produce scripts that directly implement the policy as a low-level configuration [Couch and Gilfix, 1999]. However, they conceded that interpreting Prolog code can be difficult for inexperienced users, and that they did not view Prolog as a language for administrators to use directly. Ultimately they advocated the creation of a tool which generates Prolog code from some other policy language.

Configuration tools which incorporated Prolog soon emerged. Narain *et al.* created a "service grammar" which allows the composition of lower-layer protocols while preserving end-to-end security requirements [Narain et al., 2003]. The service rules are implemented in Prolog, as high-level constraints describing valid configurations of components in a network. The system is capable of generating valid configurations given a set of components, as well as identifying inconsistencies or errors in existing configurations.

Prototype system configuration tools developed at HP Labs also utilised logic programming. An initial attempt attached policies to CIM classes using a modified version of the object-oriented SmartFrog language and a custom logic solver [Sahai et al., 2004b]. Subsequently this model was extended with "activities" attached to each CIM class, which are used to create a workflow for the implementation of the generated configuration [Sahai et al., 2004a]. A final prototype expresses constraints using an extended version of the MOF file format used to describe CIM models [Hinrich et al., 2004]. Although this final paper claims to formulate the problem as an Object-Oriented Constraint Satisfaction Problem (CSP), its constraints are expressed as first-order logic and solved with the EPILOG first-order resolution solver. Indeed, the authors note that using this methodology it is possible to express constraints which are undecidable, which would not be the case if the problem was represented as a finite domain CSP.

Logical constraints were added to the SmartFrog configuration language, which is publicly available from HP Labs [Company, 2005]. The constraints are embedded within the SmartFrog policy as literal strings of Prolog, delimited by `#suchThat#`.

Rightfully the authors note that this implementation leaves much to be desired in terms of usability. Constraint satisfaction is provided by the ECLiPSe solver, which uses a mixture of constraint and logic programming. One notable feature of SmartFrog's adoption of constraints is that they can be evaluated at run-time, in order to assign values to variables at the time the policy is implemented on a client machine. However, this means that the precise behaviour of the policy cannot be verified beforehand.

3.1.3 Boolean satisfiability (SAT)

Boolean satisfiability (SAT) is the problem of finding an assignment of variables in a boolean formula which make the formula true. Compared with logic programming, SAT is a highly restricted problem-solving method, however it has the advantage of being decidable, and typically simple enough to solve efficiently.

Alloy

The Alloy Analyzer [Jackson, 2002] is a SAT-based relational modeling system. Alloy provides a language with relational and logical constraints, and given a bounded search-space can find either instances or counterexamples of the model. Its design is deliberately simple, focusing on only those elements which are amenable to automatic analysis, and phrasing all data structuring with the notion of a relation. Alloy has relatively poor support for sequences and integers [Jackson, 2012], which can be explained by their cumbersome SAT encodings. While Alloy is a capable model finder, it does not support the finding of optimal models, a significantly more complex task.

Network configurations have been automatically generated by directly modelling the problem in Alloy [Narain, 2005]. However, this approach did not scale to networks of a realistic size. Subsequently, Prolog was used to express constraints, and those which Prolog could not solve directly were reduced to quantifier-free form and passed directly to a solver [Narain et al., 2008]. The solver used was the Kodkod relational logic solver which utilises a SAT solver and also serves as the backend to Alloy [Torlak and Jackson, 2007]. Partial evaluation of the problem in Prolog results in far fewer constraints being passed to the solver, and improved run times. However, the approach is only capable of solving boolean combinations of simple arithmetic constraints on integers, which the authors note is restrictive and they advocate the replacement of the SAT solver with a more powerful SMT solver as possible future work.

Kodkod

The Margrave tool for firewall analysis makes use of the Kodkod solver [Nelson, Barratt, Dougherty, Fisler, and Krishnamurthi, 2010]. Margrave is able to take a Cisco router configuration file and validate it, detecting overlaps and conflicts amongst rules, and identifying the consequences of configuration edits. While Margrave's performance is acceptable, the authors note that it is slower than other more simple firewall analysers and will scale poorly to large network sizes.

MiniSat

Engage is a prototype deployment system which makes use of the MiniSat SAT solver to solve dependency constraints between components [Fischer, Majumdar, and Esmaeilsabzali, 2012]. It has a small, formalised type system with component subtyping, though it does not include algebraic constraints. Deployments can be made to Rackspace or Amazon Web Services. The configuration language is a subset of JSON and focuses on describing resource structure and dependencies. Constraints are over versioned dependencies, with greater than, less than, and equality constraints supported. Resources map onto a real-world entity such as "MySQL", with a "driver" written in Python to control the resource. This is a similar approach to that seen in CFEngine and Puppet.

3.1.4 Satisfiability modulo theories (SMT)

The satisfiability modulo theories (SMT) problem is a generalisation of the boolean SAT problem, in which some variables are replaced with predicates from a range of other theories; typically integers, real numbers, arrays, and bit vectors. This extends the expressiveness of the SAT problem significantly, though it can introduce undecidable problems. The motivation for SMT is performance, for example the SAT encoding of integers is particularly cumbersome, yet under SMT a linear solver could be used to quickly solve algebraic constraints.

Cauldron

Cauldron represents the state-of-the-art in declarative configuration [Ramshaw, Sahai, Saxe, and Singhal, 2006]. A research prototype, developed at HP Labs, it makes use of an SMT solver to automatically generate valid configurations from an object-oriented,

declarative policy language.

Cauldron's policy language incorporates object-oriented concepts from Java and CIM, and is intended to be usable directly by a system administrator or designer. The language includes classes, objects and primitives, arrays, single inheritance, and object references, a well as a logical expression language for constraints. An example Cauldron model is shown in Figure 3.2, which is taken from [Ramshaw, Sahai, Saxe, and Singhal, 2006].

Arrays in Cauldron may be of variable size, with a fixed upper bound, in which case the solver is free to choose the cardinality of the set within the bound. Quantifiers may be placed over array elements, but because of the upper bound this does not result in search becoming incomplete. In Cauldron quantifiers are known as "group operators" and take the form $Q(id, arr, expr)$, where Q is either **gand** (group and) or **gor** (group or)—which are the universal and existential quantifiers, respectively—*id* is an identifier, *arr* is an array, and *expr* is a boolean expression.

Reference types in Cauldron act like pointers to objects, except that the object to which they refer is decided by the solver. References cannot be null and are constrained to a specific type, `ref Server` must refer to a `Server` instance somewhere in the solution.

Cauldron translates an object-oriented policy into a single first-order formula, which is passed to the VeriFun solver [Walther and Schweitzer, 2003]. VeriFun is an SMT solver, which may be used with different SAT solvers; the latest version of Cauldron uses the LazySAT solver [Singla and Domingos, 2006]. VeriFun uses an instantiation-based method to handle quantifiers, and as such its reasoning is not complete. It is possible for VeriFun to produce a solution which is actually invalid.

Each field in a Cauldron class is converted into a function which returns the value of the field given an object instance. Cauldron's `float` is actually a rational, for which VeriFun can only find solutions using a theory of rational linear arithmetic. A precise definition of Cauldron's translation process is not given, and its software implementation has not been published, which makes it difficult to reason about the semantics and limitations of Cauldron. The authors concluded their paper noting they were seeking to improve Cauldron's performance and test it on larger problems.

```
Database {
}

Computer {
  purchaseCost: float;
}

Application {
  licenseFee: float;
}

Server {
  cost: float;
  comp: Computer;
  app: Application;
  satisfy cost == comp.purchaseCost + app.licenseFee;
  satisfy cost <= 2000;
}

DataServer extends Server {
  data: Database;
}

DataCenter {
  serv: DataServer[..12];
  satisfy gand(i, serv, serv[i].cost <= 1500);

  manager: (ref DataServer)[2];
  satisfy gand(i, manager,
            gor(j, serv, manager[i] == serv[j]));
  satisfy manager[0] != manager[1];
}

main {
  dc: DataCenter;
}
```

Figure 3.2: An example Cauldron model from [Ramshaw, Sahai, Saxe, and Singhal, 2006]. With classes for Computer, Application, and DataCenter, and a single global DataCenter `dc`. A Server consists of a Computer and and Application, while a DataServer extends this with a Database. Each DataCenter contains up to 12 Servers with an individual cost of less than 1500. Two servers are designated as managers, with a constraint that the manager is chosen from among this DataCenter's servers, and that the two managers cannot be equal.

3.1.5 Generative CSP

A component-based model of configuration has been seen as desirable for some time [Mittal and Frayman, 1989]. This original vision for configuration allows unbounded dynamic creation of components and as such cannot be described by a classical CSP. The desire for a CSP formalism which allows for the introduction of variables and constraints not known *a priori* has led to the development of several variations on the CSP formalism, notably the Dynamic CSP [Mittal and Falkenhainer, 1990] and Composite CSP [Sabin and Freuder, 1996]. Neither of these formalisms account for component connections which have unbounded cardinality, which, while obviously leading to an infinite search space, is nonetheless sometimes desirable, particularly for speculative design tasks.

The problem of dynamically introduced components was addressed by a derivative of the Dynamic CSP: the Generative CSP [Stumptner and Haselböck, 1993]. It provides an object-oriented type hierarchy with inheritance, where types contain declarations of attributes, connection ports, and constraints. Both attributes and ports are strongly typed, and constraints may be applied to either. Constraints can reference neighbour components only, though it is possible to chain constraints together to navigate the component hierarchy. New components are added to the solution only when strictly required, so solutions with fewer components are preferred. The number of components is not fixed, as in a standard CSP, and do not require explicit activation as in a Dynamic or Composite CSP. To determine the size of the search space, the upper bound on the cardinality of each port is calculated with respect to its constraints. Presumably this is an overestimate, but specific details are not given. To further reduce the size of the search space, problems are decomposed into "packages" of almost-independent sub-problems. Generative CSPs were developed further at Siemens, resulting in the COCOS system used to configure large telephone exchanges, with over 40,000 components [Stumptner, Friedrich, and Haselböck, 1998].

However, the modern approach to modelling such CSPs is to describe dynamic problems as a series of classical CSPs, rather than adopting a specialised formalism. Thus there has been little response to the Generative CSP paradigm. Very few details of the COCOS system or its language are available, some fifteen years later.

3.1.6 PoDIM

PoDIM is a framework for the Eiffel language which allows constraints on objects to be described with a SQL-like syntax using an Entity-Relationship model [Delaet and Joosen, 2007]. As a configuration language it has no definition beyond its implementation as a domain-specific language for Eiffel. Subsequent efforts to combine PoDIM with an existing configuration reached the conclusion that PoDIM's implementation did not scale to problems of practical size [Delaet, Anderson, and Joosen, 2008].

3.2 Automated reconfiguration

This section discusses existing work related to automated reconfiguration. It is structured based on the underlying constraint-solving technology used in the work, starting with event-driven methods and moving on to logic programming, constraint programming, and A.I. planning.

3.2.1 Event-driven approaches

OCL, as discussed, is an event-driven approach which incorporates pre- and postconditions [Object Management Group, 2006]. As an event-driven policy language it is able to describe more than one configuration task, as subsequent tasks may be triggered by events. This is a form of reconfiguration, in which the original policy describes how the system should react to change. Within OCL postconditions it is possible to refer back to values from the previous state of the model by adding `@pre` as a postfix to any variable name, as shown in Figure 3.3. Reconfiguration with OCL suffers from the same drawbacks as configuration, primarily that the approach is imperative with respect to the system, with methods describing a sequence of actions to be taken, rather than a final goal state.

A graph-based language for reconfiguring of software architectures was proposed in [Wermelinger, Lopes, and Fiadeiro, 2001]. Reconfigurations are treated as operations on graphs, namely additions and removal of components and connections. The proposed language is procedural, with Pascal-like scripts specifying a series of actions on components with preconditions. These actions are translated into rewrite rules on the component graph. The paper is illustrative; neither the the complete translation process nor formal definitions are provided.

```
context Machine::addDisk(size : Integer) : Integer
  post addDiskPost:
    totalSize = totalSize@pre + size
```

Figure 3.3: An OCL postcondition, in which the `Machine` class' `totalSize` is incremented after the `addDisk` method is called. The `@pre` postfix provides access to the previous value of a variable, in this case `totalSize@pre`.

3.2.2 Logic programming

Rhizoma is an experimental distributed software deployment system which utilises constraints [Yin, Cappos, Baumann, and Roscoe, 2008]. It is capable of generating configurations given a set of resources and constraints over them, both described in Prolog. The ECLiPSe constraint-logic-programming system is used to solve constraints, using a centrally-stored knowledge base reflecting the system state. Decision-making is made only on an elected "leader" node, though any node is capable of becoming the leader. Optimisation plays an important role in Rhizoma. The ability of ECLiPSe to maximise a utility function is used to configure a distributed service to maximise its performance constraints, and take into account the cost of acquiring and releasing virtual machines. Configurations do not consist of a declarative goal state, but instead a set of "moves" which improve the service's state. Starting from the current system state, Rhizoma makes it possible to try to limit the number of moves in order to obtain a lower-impact reconfiguration. However, limiting the number of moves does not guarantee a minimal-impact reconfiguration when counteracted by an objective function.

In [Castaldi, Costantini, Gentile, and Tocchio, 2004], the DALI multi-agent system is combined with Lira, a network-based reconfiguration system. The combination of DALI's Prolog-based reasoning with Lira's Java-based networking protocols allows global reconfigurations to be performed dynamically through the cooperation of the agents. Reconfiguration tasks are encoded as a set of action rules with preconditions. Like Rhizoma, the use of rules with preconditions makes this an action policy language, despite the fact that individual logical constraints are expressed declaratively.

3.2.3 Constraint programming (CP)

Virtual machine allocation using a constraint programming (CP) solver was studied as the bin-repacking scheduling problem in [Hermenier, Demassey, and Lorca, 2011]. This problem starts with a sub-optimal placement of virtual machines onto physical machines, perhaps due to a failure occurring. The virtual machines are then reassigned, with the number of moves being minimised. The output of the constraint program is a series of "moves", hence it is both a bin-packing and scheduling problem. The problem was encoded as a constraint program using the Choco CSP solving library [Jussien, Rochart, Lorca, et al., 2008]. The solver was able to solve models with up to 2,000 servers and 10,000 virtual machines in around 220 seconds, using a heuristic "repair mode" in which some of the well-placed virtual machines were *a priori* fixed to their current location. The results were incorporated into the virtual machine manager Entropy [Hermenier, Lorca, Menaud, Muller, and Lawall, 2009].

3.2.4 A.I. planning

Planit combines a simple system configuration tool with an A.I. planner to perform reconfiguration tasks [Arshad, Heimbigner, and Wolf, 2003]. Planning constraints are written in the standard PDDL planning language, and solved with the LPG planner [Gerevini and Serina, 2002]. Planit has a built-in model of machines and components which may run upon them. Reconfiguration is performed after component failure by incorporating the non-failed components into the goal state used by the planner, and updating the initial state to match the system. This means that reconfigurations which would require moving a non-failed component are not possible.

3.2.5 Local search

The 2011–2012 Google/ROADEF challenge covered a machine reassignment task at large-scale [ROADEF, 2011]. The goal was to improve the usage of a set of machines, which each had a number of processes assigned to them. Scheduling was not taken into account, instead the goal was to generate a better static configuration within five minutes. Hard constraints included limits on the capacity of each machine versus the resources required by each process, certain processes which conflict with each other, and dependencies between processes. Soft constraints included minimising the percentage load of each machine, the balancing of work across available machines,

and the cost of moving a process, which varies between machines.

The scale of the problem was large: up to 5,000 machines and 50,000 processes. The winning entry was a custom-coded local search method, which iteratively tries to replace the current solution with an improved one [Gavranović, Buljubašić, and Demirović, 2012]. Heuristics were devised specifically for the problem, and the search progresses by performing problem-specific moves such as swapping pairs of virtual machines. Such local search methods are not complete, as they can become trapped in local minima, however we can see that when tailored well, the results can be impressive. Another downside to this approach is that expert knowledge is required to craft the solving strategy and heuristics, and the solver is limited to solving exactly one specific problem.

3.2.6 SmartFrog & LCFG

The SmartFrog and LCFG configurations tools were combined in [Anderson, Goldsack, and Paterson, 2003] to create a prototype tool capable of exploiting SmartFrog's component-based peer-to-peer orchestration with LCFG's low-level system configuring abilities. Although the system is able to respond to change, its logic to do so is custom-coded in Java for each component.

3.2.7 Dynamic software updating

Dynamic Software Updating (DSU) is a formalised methodology in which C programs may be updated at runtime, [Hicks and Nettles, 2005]. DSU provides the ability to infer a runtime patch based on changes between source files, in a manner similar to the Unix `diff` utility. Some changes ultimately require manual intervention, but the approach taken to automating patch generation is interesting due to the fact that changes between the new and old program are inferred.

3.3 Constraint modeling languages

This section discusses recent research into modelling languages for constraint programming (CP). Such languages build on top of the CSP formulation of the problem, in terms of variables, domains, and constraints. This thesis makes use of once such language, MiniZinc which is discussed in section 2.2.3.

The Zinc language, of which MiniZinc is a subset, is a rich solver-independent modelling language [Marriott, Nethercote, Rafeh, Stuckey, De La Banda, and Wallace, 2008]. Zinc incorporates far more functionality than MiniZinc, including features such as coercions via unification, both continuous and discrete domains, records types, tuples, enumerations, and fixed-size nestable sets and arrays. However, in practice only the G12 solver supports Zinc, and it is missing support for many key features [Stuckey, de la Banda, Maher, Marriott, Slaney, Somogyi, Wallace, and Walsh, 2005]. Nor has a compiler capable of translating Zinc to MiniZinc been developed.

Essence is a solver-independent modelling language based around natural language and discrete mathematics [Frisch, Grum, Jefferson, Hernández, and Miguel, 2007]. It aims to provide a familiar interface to those outside the CP community who are familiar with discrete mathematics. Essence is less sophisticated than Zinc, lacking user-defined predicates, functions, or constrained types, but it is still ambitious and far more complex than MiniZinc. Its features are mathematically inspired: nestable sets, multi-sets, relations, partitions, and functions. Like Zinc, implementations of Essence have suffered from the inherent complexity of the language and the mismatch between what can be expressed and what current CSP solvers can solve. Indeed, it took four years for the first full prototype compiler for Essence to appear [Akgun, Miguel, Jefferson, Frisch, and Hnich, 2011], and it is compatible with only a single CSP solver.

s-COMMA is an object-oriented constraint modelling language which provides classes, subtyping, and conditional components. A conditional component is field with a constraint which determines whether or not that field's object should be instantiated. s-COMMA does not include sets or arrays, which greatly simplifies the semantics of conditional components. Nor does it support quantification, or object references as seen in Cauldron (see section 3.1.4). s-COMMA interfaces directly with a number of constraint solvers such as Gecode/J and ECLiPSe.

Chapter 4

ConfSolve by Example

This chapter provides an overview of the ConfSolve language developed in this thesis. This informal description is intended to demonstrate each of the language features in a practical manner.

4.1 Language Features

ConfSolve provides the user with an object-oriented declarative language, with a Java-like syntax, which adheres to several key principles:

1. Order never matters. Declaration and usage can occur in any order with no difference in meaning.
2. Everything is an expression, except declarations.
3. All classes are equal: there are no built-in classes with special meanings such as Machine or File.

4.2 Variables and Classes

A ConfSolve model consists of a global scope in which strongly-typed variables, classes, and enumerations may be declared. For example, a simple machine may be defined as:

```
enum OperatingSystem { Windows, UNIX, OSX }

class Machine {
  var os as OperatingSystem;
  var cpus as 1..4;
  var memory as int;
}

var m0 as Machine;
```

In which `m0` is a Machine object in the global scope, with members `os`, an enumeration; `cpus`, an integer subrange; and `memory`, an unbounded integer. This declaration of a variable of type Machine results in a new Machine object being statically allocated.

Member variables may also declare objects, allowing the nesting of child objects within a parent object. For example, we could add a network interface to the machine definition:

```
class Machine {
  ...
  var en0 as NetworkInterface;
}

class NetworkInterface {
  var subnet as 0..3;
}
```

An instance of NetworkInterface will be created whenever a Machine is instantiated. The lifetime of the NetworkInterface instance is tied to that of its parent object, and is not shared between different instances of Machine.

4.3 Inheritance

Objects support classical single inheritance via abstract classes. For example, we declare a class model machine-roles, with specialised subclasses for web servers:

```
abstract class Role {
  var machine as ref Machine;
}

class WebServer extends Role {
  var port as 0..65535;
}
```

4.4 References

Associations between objects are modelled using reference types. References are handles to objects elsewhere in the model, which cannot be null. By default, object declaration is instantiation. Consider an instance of the web server role:

```
var ws1 as WebServer;
```

In the previous declaration of the Role class, the variable `machine` was declared as a Machine reference. Thus `ws1` contains a reference to a machine, in this case it will refer to `m0`, as it is the only machine we have so far declared. The solver will automatically assign the value of a reference to any instance of the appropriate type, so if we always wanted `ws1` to run on `m1` we would also need to write:

```
ws1.machine = m1;
```

Which is an example of an equality constraint.

4.5 Constraints

Constraints are expressions which must hold in any solution to the model. For example, introducing a database-server role which can be either a slave or master, and must be peered with another slave or master, as appropriate:

```
enum DatabaseRole { Master, Slave }

class DatabaseServer extends Role {
  var role as DatabaseRole;

  // slave or master
  var peer as ref DatabaseServer;

  // the peer cannot be itself
  peer != this;

  // a master's peer must be a slave,
  // and a slave's peer must be a master
  role != peer.role;
}
```

This allow us to define two database server roles:

```
var masterDB as DatabaseServer;
masterDB.role = DatabaseRole.Master;
masterDB.peer = slaveDB;

var slaveDB as DatabaseServer;
slaveDB.role = DatabaseRole.Slave;
slaveDB.peer = masterDB;
```

Likewise we may define logical boot-disks on a SAN for each physical machine, and assign logical boot-disks to the two roles:

```
var db_disk as LogicalDisk;
db_disk.capacityGB = 2048;

var web_disk as LogicalDisk;
web_disk.capacityGB = 10;
```

Updating the declarations of Machine, WebServer and DatabaseServer with:

```
class Machine {
  ...
  var bootDisk as ref LogicalDisk;
}

class WebServer extends Role {
  ...
  machine.bootDisk = web_disk;
}

class DatabaseServer extends Role {
  ...
  machine.bootDisk = db_disk;
}
```

4.6 Sets and quantifiers

Sets of variables may be declared, for example 10 web servers:

```
var webServers as WebServer[10];
```

A quantified constraint over the members of `webServers` can ensure that each server's port is set to 80, as long as the role is not running on `m0`:

```
forall ws in webServers where ws.machine != m0 {
  ws.port = 80;
};
```

As the port of `m0` is not constrained, the solver is free to choose its value. Should we want to specify it ourselves, we could write:

```
m0.port = 443;
```

So far our model contains only one Machine, `m0`, let's declare a class to describe a rack of 24 machines:

```
class Rack {
  var machines as Machine[24];
}

var r1 as Rack;
var r2 as Rack;
```

Here `machines` is declared as a set of objects, 24 new instances of Machine will be created as children of each Rack instance, in this case `r1`.

Given the following constraints, which place the master and slave databases in different racks:

```
masterDB.machine in r1;
slaveDB.machine in r2;
```

If rack `r2` fails, is there a valid solution? The answer is clearly no, and we can perform a quick impact analysis of such a failure by simply commenting out `r2`. Alternatively we could modify the definition of a Role to be:

```
abstract class Role {
  var machine as ref Machine;
  machine in r2.machines = false;
}
```

In either case ConfSolve will report that there is no valid solution to the model.

4.7 Optimisation

Minimisation and maximisation constraints may be used for any solver-populated variable. The solver will find not just a valid value, but an optimal value, given some expression to be maximised or minimised. For example, if we prefer database masters and slaves to be in different racks, but this is not a hard constraint, then we can remove the constraints:

```
masterDB.machine in r1;
slaveDB.machine in r2;
```

Replacing them with a constraint maximising the number of machines with peers on different racks:

```
var databases as ref DatabaseServer[2];
var racks as ref Rack[2];

maximize sum r in racks {
  count (db in databases
    where db.machine in r.machines
      != db.peer.machine in r.machines);
};
```

4.8 Output

The final output of the ConfSolve compiler, once solving is complete, is an object-tree in a format similar to the popular JavaScript Object Notation (JSON) format, which we call ConfSolve Output Notation (CSON). We describe CSON in full in Section 5.4.6 and explain the rationale for its design. As an illustrative example, the CSON corresponding to the model containing only `m0` is as follows:

```
{
  m0: Machine {
    os: OperatingSystem.UNIX,
    cpus: int 4,
    memory: int 1024,
    en0: NetworkInterface {
      subnet: 0,
    },
```

```
    bootDisk: ref LogicalDisk web_disk,
  },
  web_disk: LogicalDisk {
    capacityGB: int 10,
  },
}
```

4.9 Reconfiguration

This completes the description of the features of the basic ConfSolve language, the formal description of which is given in Chapter 5. There is more to the ConfSolve language though, in Chapter 7 we develop a series of extensions which facilitate reconfiguration tasks.

Chapter 5

The Syntax and Semantics of ConfSolve

In this chapter we define the abstract grammar of the ConfSolve language and its output format, and define its translation to a constraint satisfaction problem encoded in MiniZinc. This chapter is based on work from [Hewson, Anderson, and Gordon, 2012] and [Hewson and Anderson, 2011].

5.1 Core syntax of ConfSolve

This section describes the abstract grammar of ConfSolve, which is independent of the concrete grammar which we chose for our implementation, and does not include concrete syntax such as semicolons, comments, or whitespace rules. This provides a concise description of the language, free from unnecessary detail. It also allows others to use their own concrete syntax, but adopt the ConfSolve abstract syntax tree (AST) in order to apply the same translation steps to target MiniZinc.

To avoid redundancy, we first define a minimal core language, and then a series of derived constructs which are defined in terms of the core language.

Syntax of types:

$T ::=$	type	
int		integer
t		element type
$t[]$		set of t
$c[n]$		set of objects, with cardinality n

$t ::=$	element type
bool	boolean
S	integer subset
u	enumeration
c	object
$S ::= \{i_1, \ldots, i_n\}$	integer subset

Identifiers are represented by metavariables: c is a class name, v is a variable name, u is an enum name, a_i is an enum member, l is a field name; i, m and n are integers; and b is a boolean: **true** or **false**.

A ConfSolve type is either a boolean, and unbounded integer, an finite subset of integers, an enumeration, and object, a set of element types, or a set of objects with a fixed cardinality. Nesting of set types is not supported, as there is no direct way to represent sets of sets in MiniZinc, however an object may contain a set field, which may itself contain objects, giving the user the means to model arbitrary nesting via objects. Variable declarations may not be of type $c[]$, which is reserved for use during type checking (see section 5.2).

Integers are the only unbounded type in ConfSolve, as is the case in MiniZinc. Consequently a set of **int** cannot be declared, a restriction which we formally impose when we describe the type system in section 5.2. This restriction stems from the fact that quantifiers are unrolled as part of the MiniZinc to FlatZinc compilation process, which is not possible when the domain of the quantifier is infinite. As it is usually undesirable to have unbounded models, it is worth observing that the benefit of the **int** type is that it allows constants whose domain is not known to be declared and assigned separately. Thus one can define **var** *id* **as int** and later write the constraint $id = 4$. It also allows the user to avoid having to specify the domain of functionally defined variable values which ultimately depend on only variables with finite domains, for example the domain of $x = 5 * y + 3$, where y is a constant defined elsewhere.

To reduce the complexity of the MiniZinc encoding, sets of objects $c[n]$ have the same upper and lower bound n on their cardinality. As an alternative, the user may instead use a fixed cardinality set of objects, and a variable cardinality set of references with a constraint that the latter must be resolved to only members of the former. The derived expressions in Section 5.1.1 address the declaration of such fixed-cardinality sets.

Syntax of expressions:

e ::=	expression
this	current object
v	variable
$e.l$	field access
$u.a$	enum member
e.**size**	set cardinality
e_1 BinOp e_2	binary operator
Fold (v **in** e_1 **where** e_2) (e_3)	fold
bool2int(e)	cast bool to int
-e	arithmetic negation
!e	logical not
$[e_1, \ldots, e_n]$	set literal
b	boolean literal
i	integer literal
(e)	parenthesis
Fold ::=	fold operator
forall | **exists**	quantification
sum	summation

Variables, constants, binary and unary operators, and parenthetical expressions are defined in the standard manner. Object field access $e.l$ evaluates to the field l of object e. Enum constants are written in a fully-qualified manner as $u.a$, where u is the name of the enumeration and a is a constituent member. The current object can be accessed via **this** within the body of a ClassDecl. For expressions with set-type, e.**size** evaluates to the cardinality of the set given by e. Three folds over sets are defined: universal quantification, **forall**, existential quantification, **exists**, and summation, **sum**. Folds include a **where** expression which filters the set prior to evaluating the fold. Finally, the function **bool2int** provides type-casting between boolean and integer types.

Binary operators:

BinOp ::=	binary operator
= | > | >= | < | <= | **in**	relational
union | **intersection** | **subset**	set
&& | || | -> | <->	logical
+ | - | / | * | ^ | **mod**	arithmetic

Relational operators use the standard C-like notation, with the addition of **in** which is the set membership operator $\in$, and **subset** which is the subset operator $\subset$. Logical operators are *and*, *or*, *implies*, and *biconditional*. Arithmetic operators are standard, where ^ is exponentiation, and **mod** is modulo.

Syntax of models:

Model ::=	model
Declaration*	declarations
Declaration ::=	declaration
ClassDecl	class decl.
EnumDecl	enum decl.
VarDecl	var decl.
Constraint	constraint
ClassDecl ::=	class decl.
abstract? **class** c **extends** c' {	
(VarDecl | Constraint)*	
}	
EnumDecl ::=	enum decl.
enum u $\{a_1, \ldots, a_n\}$	
VarDecl ::=	variable decl.
var v **as** T	
var v **as ref** c	object reference
Constraint ::=	constraint
e	hard constraint
maximize e	soft constraint

Identifiers are represented by metavariables: c is a class name, v is a variable name, u is an enum name, a_i is an enum member, l is a field name; i, m and n are integers.

A model consists of a series of declarations, of either classes, enumerations, variables, or constraints. A class declaration may contain any number of nested variable or constraint declarations, and it may extend another class.

A declaration **class** c **extends** c' is well-formed if and only if, there is a well-formed class declaration for c' and the inheritance hierarchy is acyclic, or if c' is the top class, denoted by the distinguished name $\varnothing$. A well-formed class may contain duplicate field names.

Enumerations consist of a name and a non-empty a set of identifiers, which defines its members. Variables are always declared with a type T. Object reference variables

may be declared using **var** v **as ref** c. This creates a reference which will resolve at solve-time to an instance of c elsewhere in the model, whereas the declaration **var** v **as** c allocates a new instance of c.

5.1.1 Derived syntax

The grammar above describes the core of the ConfSolve language. The full language contains a number of constructs which are derived from the core language, to keep the core and its translation as simple as possible. These syntactic re-write rules are performed by the parser in our implementation. The relation $\triangleq$ means "equal by definition".

Derived declarations:

class c { (VarDecl | Constraint)* } $\triangleq$
  class c **extends** $\varnothing$ { (VarDecl | Constraint)* }

var v **as** m **..** $n \triangleq$ **var** v **as** $\{x \mid x \in \mathbb{N},\ x \geq m \wedge x \leq n\}$

var v **as** $t[m$ **..** $n] \triangleq$
  var v **as** $t[]$
  v**.size >=** m && v**.size <=** n

minimize $e \triangleq$ **maximize** $-e$

Classes without a base type extend the top class, denoted by the distinguished name $\varnothing$. Integer subsets may be declared as ranges. Variable cardinality sets are given a shorthand notation. Minimisation is defined as negated maximisation.

Derived expressions:

Fold $(x$ **in** $e_1)$ $(e_2) \triangleq$ Fold $(x$ **in** e_1 **where true**$)$ (e_2)

count $(x$ **in** e_1 **where** $e_2) \triangleq$ **sum** $(x$ **in** e_1 **where** $e_2)$ (1)

count $(x$ **in** $e_1) \triangleq$ **count** $(x$ **in** e_1 **where true**$)$

$e_1 != e_2 \triangleq !(e_1 = e_2)$

$\{e_1; \ldots; e_n\} \triangleq (e_1 \wedge \cdots \wedge e_n)$

Quantifiers without filters are defined as having an always-true filter. The body of a **forall** expression is defined as the logical conjunction of its sub-expressions. The **count** expression is defined in terms of summation. The "not equals" operator is defined as negated equality. Finally, a semicolon-delimited expression block is defined as the conjunction of its sub-expressions.

5.2 Type system

From the information presented so far, we are able to recognise a syntactically correct ConfSolve program. However, not all syntactically correct ConfSolve programs are well-formed. For example, a program which compares booleans with integers, declares a set of **int**, or makes use of undeclared variables. To complete our description of ConfSolve, it is necessary to describe its type system.

Static typing serves two purposes, firstly to provide a level of compile-time safety, and secondly to satisfy the CSP solver's requirement that a domain is specified for each variable. The smaller the domain, the better performance one can expect from the solver. This is why the type 1..3 is more desirable than the type **int**.

We formally specify ConfSolve's type system as a *proof system*, which is a declarative specification of the rules governing the assignment of types to expressions. This is separate from the actual type checking algorithm used in the ConfSolve compiler which implements these rules.

Within a proof system types are assigned to expressions via typing *judgements*, which are applied recursively, and take the form:

$$\frac{\textit{premises}}{\textit{conclusion}}\ (\text{Name})$$

where premises may be written on multiple lines or separated with a space. Judgements which contain expressions require a *typing environment* to resolve variable names to their declared type, which is similar to the concept of a symbol table, used when implementing such systems.

Environments:

$E ::= v_1 : T_1, \ldots, v_n : T_n$	type environments
$dom(v_1 : T_1, \ldots, v_n : T_n) = \{v_1, \ldots v_n\}$	environment domain

We now begin the formal specification. Type system judgements may be made with respect to a typing environment E, of the form $v_1 : T_1, \ldots, v_n : T_n$, which assigns a type to each in-scope variable. We write $\varnothing$ for the initial environment with an empty map.

Typing judgements:

$E \vdash \diamond$	environment E is well-formed
$\vdash T$	the type T is well-formed
$T <: T'$	type T is a subtype of T'
$E \vdash e : T$	in E, expression e has type T

There are four judgements which we may make. That an environment is well formed, that a type is well-formed, that a type is a subtype of another, and that an expression has a given type.

Rules of well-formed environments and types:

Where D_c is the set of class declarations and D_e is the set of enum declarations.

(Env Empty)

$$\frac{}{\varnothing \vdash \diamond}$$

(Env Var)

$$\frac{E \vdash \diamond \quad \vdash T \quad v \notin dom(E)}{E, v : T \vdash \diamond}$$

(Type Bool)

$$\frac{}{\vdash \textbf{bool}}$$

(Type Int)

$$\frac{}{\vdash \textbf{int}}$$

(Type Int Sub)

$$\frac{}{\vdash S}$$

(Type Enum)

$$\frac{\textbf{enum}\ u\ \{\, a_i{}^{i \in 1..n} \,\} \in D_e}{\vdash u}$$

(Type Obj)

$$\frac{c' \neq \varnothing \rightarrow \vdash c' \qquad \textbf{class}\ c\ \textbf{extends}\ c'\ \{\ldots\} \in D_c}{\vdash c}$$

(Type Obj Set)

$$\frac{\vdash c}{\vdash c[n]}$$

A well-formed environment is either empty, or contains a mapping of variable names to types. A well-formed type is either a bool, and int, an enum, an integer

subset, an object, a set of t, or a set of objects with a fixed cardinality. These rules and those which follow make use of definitions introduced in the syntax of types and expressions in section 5.1.

Rules of subtyping

(Extends)
$$\frac{\textbf{class } c \textbf{ extends } c' \{\ldots\}}{c <: c'}$$

(Reflex)
$$\frac{\vdash T}{T <: T}$$

(Trans)
$$\frac{T <: T' \quad T' <: T''}{T <: T''}$$

(Set Subtype)
$$\frac{t <: t'}{t[] <: t'[]}$$

(Obj n-Set Subtype)
$$\frac{c <: c'}{c[n] <: c'[n]}$$

(Obj Set Subtype)
$$\frac{\vdash c}{c[n] <: c[]}$$

(Int Sub)
$$\frac{}{S <: \textbf{int}}$$

(Int Sub Union)
$$\frac{}{S <: S \cup S'}$$

Our rules of type assignment make heavy use of subtyping for all types, not just objects, in order to make the type-assignment rules simpler. The first three rules define the familiar rules of class-based inheritance, reflexivity (that a type is a subtype of itself) and transitivity (that a subtype is a subtype of any of its supertypes). The next three rules extend this notion to sets. The final two rules define integer subsets as a subtype of **int**, and that an integer subset is a subtype of the union between that subset and another subset. Thus $\{1,2\}$ <: **int**, the purpose of which will be explained shortly.

With all of the pre-requisites in place, we can finally present the rules of type assignment for expressions:

Rules of type assignment: $E \vdash e : T$

(Subsum)
$$\frac{E \vdash e : T \quad T <: T'}{E \vdash e : T'}$$

(Var)
$$\frac{E \vdash \diamond \quad (v : T) \in E}{E \vdash v : T}$$

(Bool Const)
$$\frac{E \vdash \diamond}{E \vdash b : \textbf{bool}}$$

(Int Const)
$$\frac{E \vdash \diamond}{E \vdash i : \{i\}}$$

(Enum Const)
$$\frac{E \vdash \diamond}{E \vdash u.a : u}$$

(Set)
$$\frac{E \vdash e_i : t \quad \forall i \in 1..n}{E \vdash [e_1, \ldots, e_n] : t[]}$$

(Eq)
$$\frac{E \vdash e_1 : T \quad E \vdash e_2 : T}{E \vdash e_1 = e_2 : \textbf{bool}}$$

(Ineq Op)
$$\frac{\oplus \in \{\texttt{>}, \texttt{>=}, \texttt{<}, \texttt{<=}\} \quad E \vdash e_1 : \textbf{int} \quad E \vdash e_2 : \textbf{int}}{E \vdash e_1 \oplus e_2 : \textbf{bool}}$$

(In Op)
$$\frac{E \vdash e_1 : T \quad E \vdash e_2 : t[]}{E \vdash e_1 \textbf{ in } e_2 : \textbf{bool}}$$

(Subset Op)
$$\frac{E \vdash e_1 : t[] \quad E \vdash e_2 : t[]}{E \vdash e_1 \textbf{ subset } e_2 : \textbf{bool}}$$

(Logical Op)
$$\frac{\oplus \in \text{logical} \quad E \vdash e_1 : \textbf{bool} \quad E \vdash e_2 : \textbf{bool}}{E \vdash e_1 \oplus e_2 : \textbf{bool}}$$

(Set Op)
$$\frac{\oplus \in \{\textbf{union}, \textbf{intersection}\} \quad E \vdash e_1 : t[] \quad E \vdash e_2 : t[]}{E \vdash e_1 \oplus e_2 : t[]}$$

(Int Sub Set Op)
$$\frac{\oplus \in \{\textbf{union}, \textbf{intersection}\} \quad E \vdash e_1 : S_1[] \quad E \vdash e_2 : S_2[]}{E \vdash e_1 \oplus e_2 : (S_1 \oplus S_2)[]}$$

(Arith Op Int)
$$\frac{\oplus \in \text{arithmetic} \quad E \vdash e_1 : \textbf{int} \quad E \vdash e_2 : \textbf{int}}{E \vdash e_1 \oplus e_2 : \textbf{int}}$$

(Arith Op Int Sub)
$$\frac{\oplus \in \text{arithmetic} \quad E \vdash e_1 : S_1 \quad E \vdash e_2 : S_2}{E \vdash e_1 \oplus e_2 : \{x_1 \oplus x_2 \mid x_1 \in S_1,\ x_2 \in S_2\}}$$

(Dot)
$$\frac{E \vdash e : c \quad c <: c' \quad j \in 1..n \quad \textbf{class } c \textbf{ extends } c' \{ \textbf{var } l_i \textbf{ as } T_i^{i \in 1..n} \}}{E \vdash e.l_j : T_j}$$

(This)
$$\frac{E \vdash \diamond}{E \vdash \textbf{this}_c : c}$$

(Set Card)
$$\frac{E \vdash e : t[]}{E \vdash e.\textbf{size} : \textbf{int}}$$

(Channel)
$$\frac{E \vdash e : \textbf{int}}{E \vdash \textbf{int2bool}(e) : \textbf{bool}}$$

(Quant)
$$\frac{Q \in \{\textbf{forall}, \textbf{exists}\} \quad E \vdash e_1 : t[] \quad E, v : t \vdash e_2 : \textbf{bool} \quad E, v : t \vdash e_3 : \textbf{bool}}{E \vdash Q\ v \textbf{ in } e_1 \textbf{ where } e_2\ (e_3) : \textbf{bool}}$$

(Sum)
$$\frac{E \vdash e_1 : t[] \quad E, v : t \vdash e_2 : \textbf{bool} \quad E, v : t \vdash e_3 : \textbf{int}}{E \vdash \textbf{sum } v \textbf{ in } e_1 \textbf{ where } e_2\ (e_3) : \textbf{int}}$$

The first rule is that of subsumption, that an expression of a type may also take any of its supertypes. This is followed by variable resolution in the environment, and boolean integer and enumeration constants. The type of an integer constant is a singleton set containing the constant's value. This is significant, because ConfSolve forbids sets of integers, thus the set literal $[1,2,3]$ is legal in ConfSolve, having type $\{1\} \cup \{2\} \cup \{3\} = \{1,2,3\}$. Indeed, it is the reason why the integer literal 1 has type $\{1\}$, and why the type system makes an effort not to promote to integer subsets to **int** too readily.

The rules for comparisons (Eq)–(Logical Op) are relatively straightforward, and make heavy use of the subtyping rules. For example, recall that S is a subtype of **int** and is this subject to the rule (Ineq Op). Likewise, when we state that both expressions in the equality (Eq) rule must be of type T, we imply only that for both types there exists a common supertype for which they may be substituted.

The set operation (Set Op) follows a similar form, but (Int Sub Set Op) provides a specialised rule for handling integer subsets, which applies the intersection or union operator to the subset itself, as appropriate. The arithmetic operations (Arith Op Int) and (Arith Op Int Sub) follow this same pattern, with the latter being somewhat unusual. Namely, that for arithmetic operations between integer subsets, the resulting type is the integer subset containing the result of the application of the operation to all pairs in the two source subsets. The motivation for this is the same as for the (Int Const) rule, that the expression $1+2$ has type 3, and thus the set literal $[1+2]$ is legal. This is a design decision to obtain the most precise type possible for an expression, in order to reduce the domain which the solver will search for values of a given type.

The next four rules, from (Dot) to (Channel) specify types for member variable access, the **this** pseudo-variable, set cardinality, and channeling integers to booleans.

The final two rules specify types for quantification and summation expressions, which are the only rules which introduce variables into the environment, *i.e.*, the scope of v is e_2 and e_3, the **where** and body clauses, respectively.

In order to provide a succinct notation, the system has the following derived properties: if $T <: T'$ then $\vdash T$ and $\vdash T'$; and if $E \vdash e : T$ then $E \vdash \diamond$ and $\vdash T$ and $\mathit{freevars}(e) \subseteq \mathit{dom}(E)$. That is, that the subtype judgement always involves well-formed types, and that any free variables in an expression are well-formed members of its environment.

5.3 ConfSolve and MiniZinc

In this section we provide an overview of the process of compiling and solving a ConfSolve model, and of the MiniZinc constraint modelling language.

Compiling and solving a ConfSolve model requires several steps, which are illustrated in Figure 5.1; the steps are as follows:

1. The ConfSolve compiler is invoked; the model is translated into a CSP expressed in MiniZinc. This is described in Section 5.4.5.

2. The MiniZinc model is compiled into a FlatZinc model using *mzn2fzn* [Nethercote, Stuckey, Becket, Brand, Duck, and Tack, 2007].

3. The FlatZinc model is solved using a constraint solver. In our implementation we use Gecode [Gecode Team, 2006].

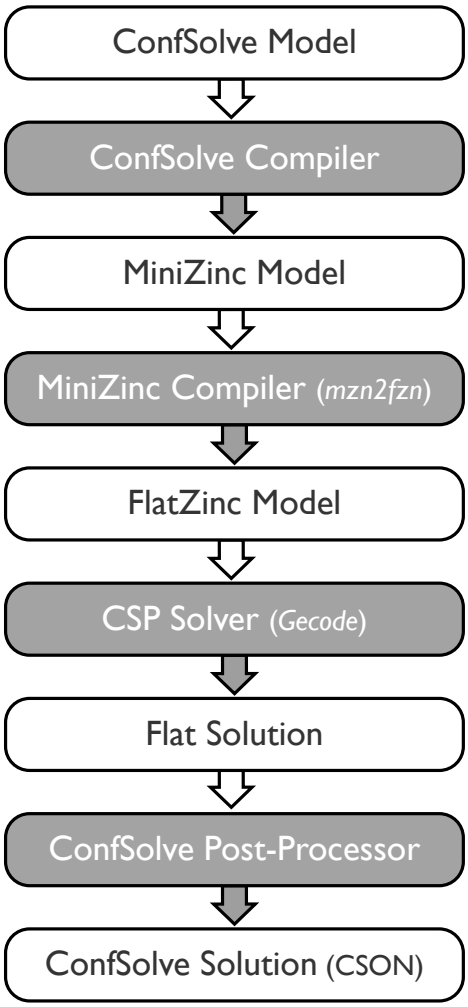

Figure 5.1: Compiling and solving a ConfSolve model. White boxes are files, shaded boxes are processes.

4. The solution found by the solver is parsed by the ConfSolve post-processor and combined with the original model to produce an object-tree (CSON) representing the solution. This is described in Section 5.4.6.

5.4 Translation to MiniZinc

This section defines the translation from a ConfSolve model to MiniZinc in terms of their abstract grammars. The translation occurs in two phases: a static allocation phase in which indexes are generated for each object and an upper-bound on the number of object instances in the model is calculated, and a translation phase in which a MiniZinc abstract syntax tree is constructed.

5.4.1 An example translation

We begin with an example translation, showing in detail a MiniZinc model generated by the ConfSolve compiler. We use the DatabaseServer model from Section 4.1, with four instances of DatabaseServer:

```
enum DatabaseRole { Master, Slave }

class DatabaseServer extends Role {
  var role as DatabaseRole;

  // slave or master
  var peer as ref DatabaseServer;

  // the peer cannot be itself
  peer != this;

  // a master's peer must be a slave,
  // and a slave's peer must be a master
  role != peer.role;
}

// instances
var master1 as DatabaseServer;
master1.role = DatabaseRole.Master;

var slave1 as DatabaseServer;
slave1.role = DatabaseRole.Slave;

var master2 as DatabaseServer;
```

```
master2.role = DatabaseRole.Master;

var slave2 as DatabaseServer;
slave2.role = DatabaseRole.Slave;
```

The resulting MiniZinc model begins with variable declarations. As MiniZinc does not support records or objects, each field in the DatabaseServer class is declared as a separate array, where the number of elements in the array is equal to the number of DatabaseServer instances in the model, in this case four. The domain of the array elements correspond to the type of the field. In the following example the DatabaseServer_role has as its domain the contiguous set of integers 1..2, which are indexes into the DatabaseRole enumeration:

```
array[1..4] of var 1..2: DatabaseServer_role;
array[1..4] of var 1..4: DatabaseServer_peer;
```

Constraints are first-order expressions which restrict the values a variable may take from its domain. Each is a boolean expression which must evaluate to true. In this case, the constraints come from the DatabaseServer class, and so are wrapped with a `forall` expression which applies the constraint to all instances of DatabaseServer, and defines the value of the variable `this` to be an index into the field arrays for the DatabaseServer class. The identifier `this` has no special meaning in MiniZinc, and was chosen simply to facilitate translation:

```
constraint
 forall (this in 1..4) (
  DatabaseServer_peer[this] != this
 );

constraint
 forall (this in 1..4) (
  DatabaseServer_role[this] !=
   DatabaseServer_role[DatabaseServer_peer[this]]
 );
```

Each MiniZinc model must have a `solve` goal, which can be to either minimise or maximise an expression, or simply satisfy the constraints in the model. In this case the goal is the latter:

```
solve satisfy;
```

Finally, the model must specify which variables values will be output by the solver. This is useful when the model includes constants or intermediate variables, the values of which are not of interest. In this case, it is simply:

```
output [
 show(DatabaseServer_role),
 show(DatabaseServer_peer)
];
```

5.4.2 Quantifiers

ConfSolve does not place any restrictions on quantifiers. However, the ConfSolve type system limits all sets to finite domains, thus sets of **int** are forbidden, and therefore so is any non-finite quantification. This is necessary to ensure decidability; quantification in MiniZinc must be finite for the same reason. The MiniZinc compiler unrolls quantified expressions at compile-time, which means that nested quantifiers can cause an explosion in the size of the generated FlatZinc model. A nested quantification over two set of m and n elements, respectively, generates $m \times n$ unrolled sub-expressions. We have been satisfied with the performance of large models, but there is necessarily a limit to their scaling, which we examine in Chapter 6.

5.4.3 Correctness

Our translation from ConfSolve to MiniZinc aims to satisfy a number of correctness properties. Firstly, that a well-formed ConfSolve model should yield a well-formed MiniZinc model; that is, a MiniZinc model free of invalid syntax or invalidly typed expressions and declarations. Secondly, a well-formed ConfSolve type should yield a well-formed MiniZinc domain. Thirdly, a well-formed ConfSolve expression should yield well-formed MiniZinc expression, in which there are no type errors. Finally, the MiniZinc model should be equivalent to the ConfSolve model. We define this equivalence in terms of solutions: a MiniZinc model is equivalent to a ConfSolve model when it produces equivalent solutions. A solution to a ConfSolve model is an assignment of structured variables to values (expressed in CSON) in which all constraints in

the model are satisfied. An equivalent solution to a MiniZinc model is an assignment of flattened variables to values (expressed in FlatZinc) which can be mapped back to the original ConfSolve model to produce a valid solution.

We do not provide a formal proof of these correctness properties in our translation. However, as informal evidence for the correctness of our translation procedure we validated each of the examples used in the thesis by hand, along with many other test models designed with edge-cases in mind.

5.4.4 Static allocation

The static allocation phase determines the upper bound on the number of instances of each class, assigns each object an index, and records which indices are assigned to the subclasses of a given class. Its purpose is to generate the following two data structures, for use in the later translation phase:

count is a map from a class names c to an integer representing the count of the number of instances of c in the model.

indices is a map from a class name c to a set of integers representing the indices of each instance of c or one of its subclasses.

The values of *count* and *indices* are updated incrementally as counting progresses via the method described below. In order to handle inheritance we introduce the concept of a root class, that is, the topmost class in any given hierarchy. Formally, $root(c)$ is c when the superclass of c is $\varnothing$, otherwise it is the topmost superclass of c. The process of counting is as follows:

Given the definition **class** c **extends** c', for each global declaration **var** v **as** T:

when $T = c$, $count(root(c))$ is incremented, and its value is added to the sets $indices(c)$, and to $indices(c^*)$ for each ancestor c^* which c extends. This process is then repeated for each field **var** v **as** T declared in class c or any ancestor of c.

when $T = c[n]$, the case for $T = c$ is repeated n times.

5.4.5 Translation

The translation describes the process of generating a MiniZinc abstract syntax tree from a ConfSolve abstract syntax tree. We make use of the notation $[\![x]\!]$ to mean "the translation of x", where x is some syntactic construct, such as a type or expression.

Translation of types $[\![T]\!]$:

$$
\begin{aligned}
[\![\textbf{int}]\!] &\triangleq \textbf{int} \\
[\![\textbf{bool}]\!] &\triangleq \textbf{bool} \\
[\![\{i_1,\ldots,i_n\}]\!] &\triangleq \{i_1,\ldots,i_n\} \\
[\![u]\!] &\triangleq 1\,..\,num(u) \\
[\![c]\!] &\triangleq \begin{cases} \{\,indices(c)\,\} & \text{if } c \text{ is abstract} \quad \text{MiniZinc: } \texttt{\{x,y,z\}} \text{ is a set domain} \\ 1\,..\,count(c) & \text{otherwise} \quad \text{MiniZinc: } \texttt{1..n} \text{ is a range domain} \end{cases} \\
[\![c[n]]\!] &\triangleq \textbf{set of } [\![c]\!] \\
[\![t[\,]]\!] &\triangleq \textbf{set of } [\![t]\!]
\end{aligned}
$$

Here we define $[\![T]\!]$ to be the MiniZinc translation of a ConfSolve type T, where $num(u)$ is the number of elements in enum u. Each ConfSolve type maps directly onto a MiniZinc type, with the exception of enumerations and objects which are translated to integer indices. For set types, the translation is recursive, but only to one level, because MiniZinc does not permit sets of sets. The translation process has not yet begun: the translation of a type is used as an intermediate step in the translations which follow.

Translation of global variable declarations:

Let $index(c)$ be a non-pure function which generates a new index of class $root(c)$ by counting and returns its value. For each global declaration **var** v **as** T, introduce a declaration:

when $T = c[n]$

$[\![T]\!] : v = \{\text{for } i \in 1..n, index(c)\}$

MiniZinc: omitting the **var** keyword declares a constant

when $T = c$

$[\![T]\!] : v = index(c)$

otherwise

var $[\![T]\!] : v$

For each global declaration **var** v **as ref** c, introduce a declaration:

var $[\![c]\!] : v$

The translation phase proceeds in a similar manner to the counting phase, and makes use of object indices generated in exactly the same manner as those in the *count* map. It is important that these indices are the same, so that an index corresponds to

the correct value in the *indices* map. We define a non-pure function $index(c)$ which generates a new index of class $root(c)$ by counting, and returns its value.

Translation begins with global variable declarations, as shown above. When the type of the declared variable is a set of objects of class c with cardinality n, indices are generated for each of the n objects. When the type is a object of class c, a single index is generated. For all other types no values are assigned, and a **var** declaration is introduced.

Reference variable declarations are translated in the same manner as the "otherwise" case, that is a **var** declaration is introduced whose domain is the translation of the c type given in the translation of types above.

Translation of class-level variable declarations:

For each ClassDecl defining a class c where $count(c) > 0$, containing fields **var** f_i **as** ${T_i}^{i \in 1..n}$, introduce an array for each field f_i:
when $T_i = c'[n]$

array [1**..**$count(c)$**] of** $[\![T_i]\!] : c\_f_i =$ **[** for $i \in 1..count(c)$, **{** for $j \in 1..n, index(c')$ **}]**
when $T_i = c'$

array [1**..**$count(c)$**] of** $[\![T_i]\!] : c\_f_i =$ **[** for $i \in 1..count(c), index(c')$ **]**
otherwise

array [1**..**$count(c)$**] of var** $[\![T_i]\!] : c\_f_i$

Where class c contains fields **var** f_i **as ref** ${c_i}^{i \in 1..n}$,
introduce an array for each field f_i:

array [1**..**$count(c)$**] of var** $[\![c_i]\!] : c\_f_i$

Variable declarations nested within classes are translated by introducing an **array** which will contain the values of that field for all instances of some class c. When the type of the declared variable is a set of objects of class c' with cardinality n, an array of sets (the only nested construct permitted by MiniZinc) is declared, and for each of the $count(c)$ instances, n indices are generated. When the type is an object of class c' the declaration is simpler: an array of indices is introduced, one for each of the $count(c)$ instances. For all other types no values are assigned, and array of **var** is introduced.

Once again, reference variable declarations are translated in the same manner as the "otherwise" case, where a **var** declaration is introduced whose domain is the translation of the type c_i.

Translation of global constraints:

For each global constraint e, introduce a statement:

constraint $[\![e]\!]$

For each global constraint **maximize** e, update the objective expression o:

when o is undefined

$o = [\![e]\!]$

otherwise

$o = o + [\![e]\!]$

The translation of hard constraints consists of translating their expressions, which we discuss later. The objective expression o is the sum of every maximisation goal's expression, and is maintained throughout the translation phase. Each **maximize** constraint corresponds to a sub-expression in the objective function.

Translation of class-level constraints:

For each ClassDecl defining a class c where

$count(c) > 0$, for each constraint e, introduce a statement:

constraint forall $(\text{this } \textbf{in } 1 \textbf{..} count(c))\ ([\![e]\!])$

For each global constraint **maximize** e, update the objective expression o:

when o is undefined

$o = [\![e]\!]$

otherwise

$o = o + \textbf{sum}\ (\text{this } \textbf{in } 1 \textbf{..} count(c))\,([\![e]\!])$

Constraints may also be declared at the class-level, in which case they apply to every instance of that class. The translation occurs in the same manner as for global constraints, except that the constraints are placed over all instances in aggregate via **forall** for hard constraints and **sum** for maximisation constraints.

Translation of expressions $[\![e]\!]$:

$$
\begin{aligned}
[\![v]\!] &\triangleq \text{(see below)} \\
[\![\textbf{this}]\!] &\triangleq \text{this} \\
[\![e.l]\!] &\triangleq \mathit{classof}(e)\_l\,\textbf{[}[\![e]\!]\textbf{]} \\
[\![u.a]\!] &\triangleq \mathit{eindex}(u,a) \\
[\![e_1\ \text{Op}\ e_2]\!] &\triangleq [\![e_1]\!]\ [\![\text{Op}]\!]\ [\![e_2]\!] \\
[\![e.\textbf{size}]\!] &\triangleq \textbf{card}([\![e]\!]) \\
\text{Fold}\ (v\ \textbf{in}\ e_1\ \textbf{where}\ e_2)\ (e_3) &\triangleq \text{(see below)} \\
[\![\textbf{bool2int}(e)]\!] &\triangleq \textbf{bool2int}([\![e]\!]) \\
[\![-e]\!] &\triangleq -[\![e]\!] \\
[\![!e]\!] &\triangleq \textbf{not}\ [\![e]\!] \\
[\![[e_1,\ldots,e_n]]\!] &\triangleq \text{(see below)} \\
[\![e_1\ \hat{}\ e_2]\!] &\triangleq \textbf{pow}([\![e_1]\!],[\![e_2]\!]) \\
[\![\textbf{true}]\!] &\triangleq \textbf{true} \\
[\![\textbf{false}]\!] &\triangleq \textbf{false} \\
[\![i]\!] &\triangleq i
\end{aligned}
$$

The MiniZinc translation $[\![e]\!]$ of a ConfSolve expression e is given above, where $\mathit{eindex}(u,a)$ is the index of element a in the declaration **enum** $u\ \{a_1 \ldots a_n\}$, and $\mathit{classof}(e)$ is the identifier c when the type of e is an object of class c. Expressions are translated recursively, with the exception of literals.

The keyword **this** is translated into the identifier "this", which has no special meaning in MiniZinc. Instead, it is simply a quantified variable defined in the **forall** expression in the prior Translation of Class-Level Constraints section.

Translation of variables $[\![v]\!]$:

Within the scope of class c, the translation $[\![v]\!]$ of a variable v is:

when v is declared in class $c' \in c^*$

c'_v[this]

otherwise

v

Variables in expressions are translated in one of two ways depending on their type. If the variable occurs within the scope of class c and was declared in c' which is either

c or one of its ancestors, then the result is a lookup in the array corresponding to field v of the current instance (*i.e.*, **this**) of class c'. For example, where c' is DatabaseServer, and v is "role", the translation is:

```
DatabaseServer_role[this]
```

Otherwise, v is either a global variable or a quantified variable (from a fold), and translates directly to its identifier.

Translation of folds:

The translation of a fold expression Fold $(v$ **in** e_1 **where** $e_2)$ (e_3) is given by:

Fold $([\![v]\!]$ **in** $r)$ (b)

where $r \triangleq$

when e_1 is of type $c[n]$

$1..count(c)$

when e_1 is of type $u[]$

$1..num(u)$

when e_1 is of type $\{i_1,\ldots,i_n\}[]$

$\{i_1,\ldots,i_n\}$

when e_1 is of type **bool**$[]$

$\{$**true**,**false**$\}$

and $b \triangleq$

when Fold = **sum**

bool2int$(v$ **in** $[\![e_1]\!]$ `/\` $[\![e_2]\!]) * [\![e_3]\!]$

otherwise

v **in** $[\![e_1]\!]$ `/\` $[\![e_2]\!]$ `->` $([\![e_3]\!])$

Folds such as **foreach** translate to a similar construct in MiniZinc, but one in which the fold must be over a set of constant value, because the MiniZinc compiler unrolls the fold at compile-time. Therefore, the translation consists of a fold of an expression body b over a constant range r, which need not be contiguous.

The range r depends on the type of the expression e_1, which the fold is over. When it is a set of objects of type c, the range is the indices of all c instances. When the type is an enum, it is the valid indices of the enumeration. When the type is an integer subset the range is that subset.

Ranges are larger than one might expect. Because MiniZinc requires that ranges are constant, the range must contain all values of the relevant type, and we must correspondingly wrap the body expression e_3 with the implication, $v \in e_1 \Rightarrow e_3$, so that the constraint is placed over only members of the set in the current solution. In fact, because ConfSolve also allows a filter expression e_2, this becomes $v \in e_1 \wedge e_2 \Rightarrow e_3$.

The body expression b in fact takes two forms. For a **forall** or **exists**, the logical form just mentioned is used. For a **sum**, an arithmetic form is used: $\mathit{bool2int}(v \in e_1 \wedge e_2) \times e_3$, where *bool2int* returns a 0/1 value given a false/true boolean.

Translation of binary operators $[\![\text{BinOp}]\!]$**:**

$[\![\&\&]\!] \triangleq$ `/\`
$[\![||]\!] \triangleq$ `\/`
$[\![/]\!] \triangleq$ **div**
$[\![\text{BinOp}']\!] \triangleq \text{BinOp}'$

Binary operators are directly translated to MiniZinc operators. BinOp′ denotes all operators not explicitly listed.

Reduction of set literal expressions:

The reduction of a set literal expression $[e_1, \ldots, e_n]$ of type T is given by:
In the current scope, insert the declaration:
var set__s **as** T
Where T is a well-formed type which satisfies the (Set) typing judgement in section 5.2 and s is a unique integer.
In the current scope assume the constraint:
constraint e_1 **in** set__s $\wedge \cdots \wedge e_n$ **in** set__s
Finally, the derived expression is:
$[e_1, \ldots, e_n] \triangleq$ set__s

The translation of set literal expression is defined in terms of a reduction to a variable and associated constraints at the ConfSolve level, which should be performed before any of the other transformation steps previously listed. This reduction is necessary as it allows variables to appear inside set literals, which would otherwise not be legal in MiniZinc.

Solve statement:

when o is undefined
  solve satisfy
otherwise
  solve maximize o

The translation to MiniZinc concludes with the introduction of a solve statement, the purpose of which is to provide a criteria for the solver's search, which may be either a satisfaction of the constraints, or maximisation of an objective expression.

5.4.6 Solutions

After solving, the output of the ConfSolve post-processor is an object-tree, the syntax of which we refer to as ConfSolve Output Notation (CSON).

To obtain a solution the translated MiniZinc is compiled into FlatZinc and solved using Gecode, which outputs assignments for each variable in a simple text-based format defined by FlatZinc. Generating a CSON tree from this text is straightforward: the steps of the translation process are repeated, but whenever a MiniZinc variable would be introduced, we instead read its value from the output file, and emit the corresponding CSON representation:

Syntax of CSON:

V ::=	value
i	integer
true | **false**	boolean
$u.a$	enum member
c {Member*}	object
ref Target	object reference
$t[n]\{V_1,\ldots,V_n\}$	set literal
Member ::=	member
$v : V$	variable name : value
Target ::=	target
v	variable
Target.l	field access
Target$[i]$	set access

Each CSON value corresponds to a type in ConfSolve. The solution output consists of a single anonymous object representing the global scope. Nested within this are values for each variable. In the special case of references, the value is the fully-qualified name of the target variable, in which members of sets may be accessed via index, for example $v[i].f$ resolves to the value of field f of the i^{th} element of the set v.

CSON is not a sub-language of ConfSolve. This is because of the encoding scheme used for sets of objects in MiniZinc. The fields of a given class are translated as arrays indexed by object index. For example the array `Server_host` contains one element for each `Server` object in the model. When the model contains a set of objects, such as `Server[5]`, then their indexes will map to a contiguous range of elements in the array. However, because we have taken a set and represented it as an array, a symmetry is introduced. An ordering of objects within the set will be present in solutions to the model. This ordering must be represented CSON output for two reasons, given below.

Firstly, the ordering is necessary for the resolution of reference values. A reference is expressed as a path to some target object. In order for the path to traverse an object in a set, there must be a unique way in which to identify that object, and so CSON uses the position of the object in the translated set as its identity in the path. For example `server[3]` would be the third element of the `server` set.

Secondly, in Chapter 7 we extend ConfSolve for reconfiguration, in which a previously obtained solution is used as the starting point for subsequent configuration changes. It is therefore necessary to preserve the set ordering, otherwise the mapping between existing solution objects and model objects will be lost. CSON achieves this by annotating set elements with their index in the translated array, so that this information is available to the ConfSolve compiler when compiling a reconfiguration problem.

Chapter 6

Evaluation of Configuration with ConfSolve

In this chapter the performance of ConfSolve is measured using several system configuration problems: a large virtual machine allocation problem and several smaller problems from previous work, which we benchmark against.

The purpose of this evaluation is to demonstrate that ConfSolve can be used to model a number of different configuration problems, and is capable of successfully finding solutions to them. Furthermore, we want to ensure that ConfSolve is able to perform adequately on large models.

6.1 Experimental setup

The evaluation was performed on a machine with a 2GHz Intel Core i7 processor and 8GB of RAM, running Mac OS X version 10.8.1. We used the 64-bit MiniZinc to FlatZinc converter version 1.5.1 with the `--no-optimize` flag, and the 64-bit Gecode FlatZinc interpreter version 3.7.1.

6.2 Virtual Machine assignment

In this evaluation we use ConfSolve to generate an assignment of virtual machines to physical machines in an Infrastructure as a Service (IaaS) configuration. Each physical machine is identical, having 8 CPUs and 16GB or memory. Each virtual machine has variables representing its requirements on the physical machine resources. These declarations are as follows:

```
class Machine {
  var cpu as int;     // 1 unit = 1/2 core
  var memory as int;  // MB
  var disk as int;    // GB

  cpu = 16;           // 2x Quad Core
  memory = 16384;     // 16 GB
  disk = 2048;        // 2 TB
}

abstract class VM {
  var host as ref Machine;
  var disk as int;
  var cpu as int;
  var memory as int;
}
```

Virtual machines may be one of two sizes, large and small. Large machines have 4 CPU units, 3.5GB of memory, and 500GB of disk. Small machines have 1 CPU unit, 768MB of memory and 20GB of disk:

```
class SmallVM extends VM {
  cpu = 1;
  memory = 768;
  disk = 20;
}

class LargeVM extends VM {
  cpu = 4;
  memory = 3584;
  disk = 500;
}
```

The infrastructure consists of two racks of 48 physical machines, onto which we wish to allocate 350 small and 100 large virtual machines:

```
// physical machine instances
var rack1 as Machine[48];
var rack2 as Machine[48];

// virtual machine instances
var smallVMs as SmallVM[350];
var largeVMs as LargeVM[100];
```

We define a constraint on virtual machine placement, as otherwise there is nothing to prevent every virtual machine from having the same host:

```
var machines as ref Machine[96];
var vms as ref VM[450];

forall m in machines {
  sum vm in vms where vm.host = m {
    vm.cpu;
  } <= m.cpu;

  sum vm in vms where vm.host = m {
    r.memory;
  } <= m.memory;

  sum vm in vms where vm.host = m {
    r.disk;
  } <= m.disk;
};
```

This constraint states that for each physical machine, the sum of the required quantity of each resource over all virtual machines hosted on it, must be less than the quantity of that resource provided by the physical machine. In other words, that the virtual machines assigned do not, in aggregate, consume more resources than are available. This is repeated for the three resources, `cpu`, `memory`, and `disk`.

From this model, ConfSolve is able to automatically generate assignments of virtual machines to physical machines (PMs), by automatically finding values for the `host` variable of each `VM` instance.

Results

The performance achieved when scaling the problem up to 900 virtual machines is shown in Figure 6.1. We did not attempt higher numbers because the time to run the experiment scales highly non-linearly due to poor performance of the MiniZinc compiler. Virtual machine allocation problems in the enterprise range from 10s to 100s of machines. We are satisfied that ConfSolve is a viable configuration tool for tasks at this scale.

Surprisingly, the MiniZinc compiler is the bottleneck. We are not aware of any task which the compiler performs which has a fundamentally non-linear time bound. Indeed, the MiniZinc compilation process is fully documented in [Nethercote, 2012].

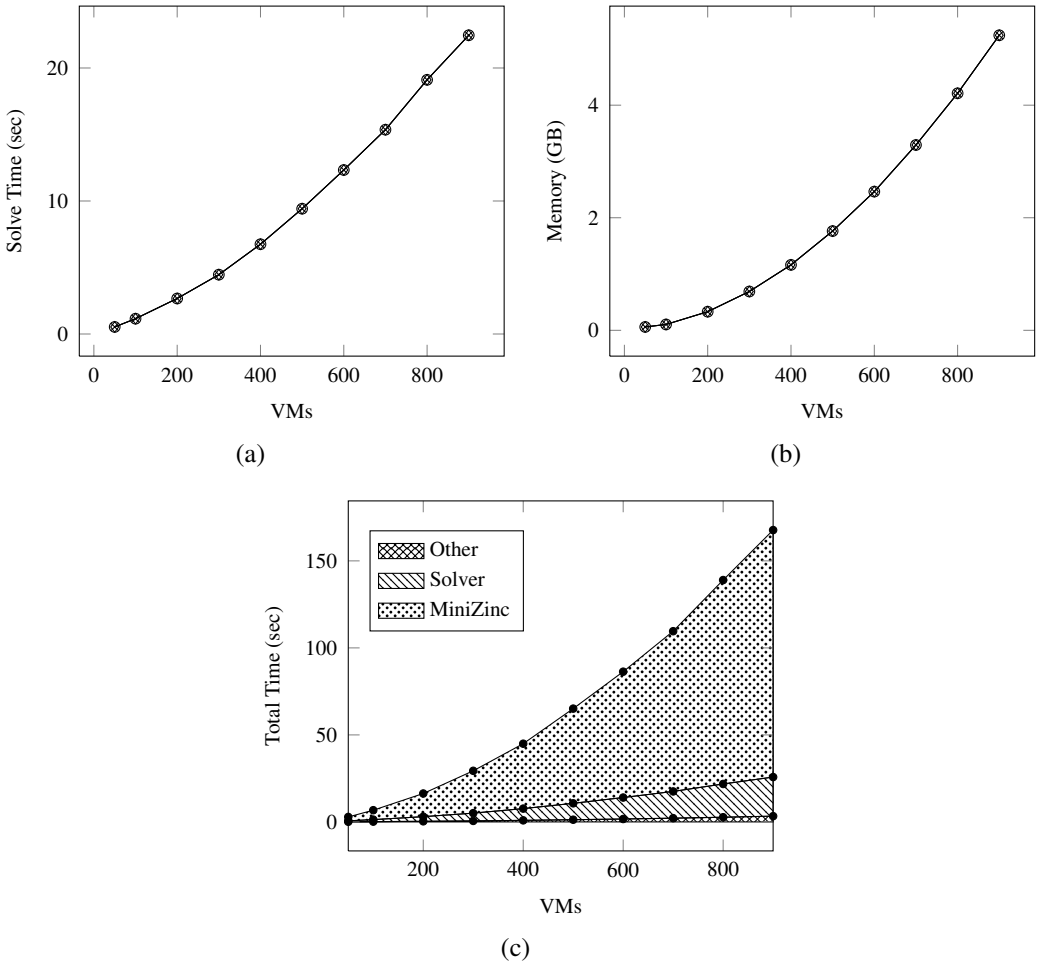

Figure 6.1: Virtual machine allocation performance. **a)** Solve time. This is the time taken for Gecode to find a solution to the FlatZinc problem. **b)** Solver memory usage, with peak usag of almost 6GB. **c)** Total time, including the time for MiniZinc compilation which, surprisingly, dominates the process. Times are disjoint. Solver referes to Gecode, and Other referes to the time taken for the ConfSolve compiler, post-processor and intermediate scripts.

Problem	ConfSolve	Cauldron
Geometry	93	327
Firewall	145	504
QM4	394	2077
ServerComplex7	652	3149
ServerComplex10	875	4254

Table 6.1: Cauldron test suite run-time in milliseconds, averaged over three runs. Shows ConfSolve consistently outperforming Cauldron.

6.3 Cauldron Test Suite

ConfSolve uses a similar object-oriented language to Cauldron [Ramshaw, Sahai, Saxe, and Singhal, 2006], a policy-based design tool which is able to describe CIM models. Cauldron is able to generate solutions to these constraint-based system designs; an example of configuring an enterprise server with physical partitions is provided in [Ramshaw, Sahai, Saxe, and Singhal, 2006].

HP Labs kindly provided us with a copy of the Cauldron binary (version `rel.10c`) and a number of sample problems from their test suite. We do not have permission to reproduce these sample problems, but the example described at length in [Ramshaw, Sahai, Saxe, and Singhal, 2006] is representative in terms of scope and size. We translated a representative subset of these problems into ConfSolve models in order to confirm that ConfSolve can represent existing constraint-based configuration models.

Results

We benchmarked the solution time of equivalent ConfSolve and Cauldron models, the results of which can be found in Table 6.1. ConfSolve consistently out-performs Cauldron by a factor of around four on the test hardware described at the beginning of this section. This difference is plotted in Figure 6.2.

The build of Cauldron which we were provided with makes use of a private build of the VeriFun theorem prover [Walther and Schweitzer, 2003]. This is used by Cauldron in conjunction with a custom SAT solver which uses the LazySAT algorithm [Singla and Domingos, 2006]. Given that Cauldron dates from 2007 and is no longer under development, more recent SAT solvers may provide a performance improvement, but Cauldron is a “black box” and we have no way to determine if this is the case.

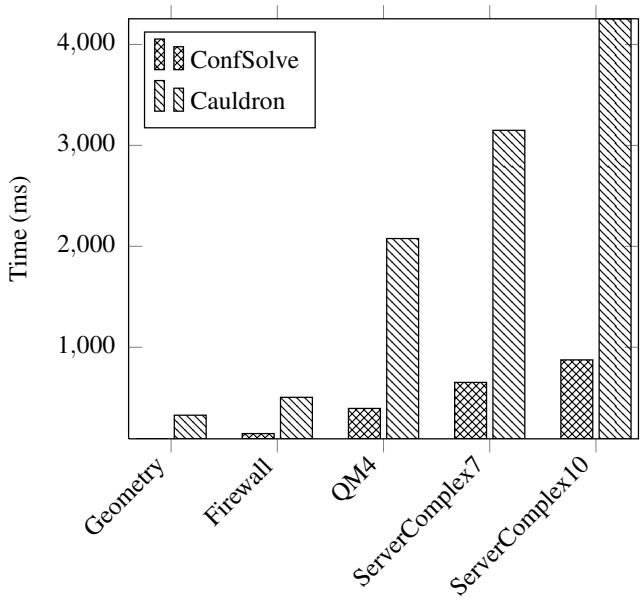

Figure 6.2: Plot of the Cauldron test suite run-time in milliseconds, averaged over three runs. ConfSolve is approximately five times faster than Cauldron.

6.4 Cauldron VM Allocation

The Cauldron test suite problems are relatively small in terms of search-space, despite some models being over 100 lines of code. We therefore translated our large-scale VM example into an equivalent Cauldron model in order to compare performance at scale.

Results

The results in Table 6.2 show that, for this example at least, Cauldron does not scale to practical problem sizes, failing when only 17 virtual machines are to be allocated to 5 physical machines. This may seem surprising at first, but Cauldron remains a prototype and problems of moderate scale are beyond its design parameters. ConfSolve, however, was able to scale to 900 virtual machines. We did not attempt higher numbers because the time to run the experiment scales highly non-linearly due to the poor performance of the MiniZinc compiler, as shown in Figure 6.1. We suspect that ConfSolve could scale much further were improvements to be made to the MiniZinc compiler, though this problem lies beyond our control.

Problem (Virtual:Physical)	ConfSolve	Cauldron
VM Allocation 4:2	151	1717
VM Allocation 8:2	165	2115
VM Allocation 16:4	177	3995
VM Allocation 17:5	194	-
VM Allocation 50:96	2774	-
VM Allocation 100:96	6739	-
VM Allocation 200:96	16,331	-
VM Allocation 700:96	109,589	-
VM Allocation 900:96	167,701	-

Table 6.2: Virtual machine allocation run-time in milliseconds, averaged over three runs. Cauldron failed to return any results beyond 17 virtual machines.

6.5 Summary

We have evaluated the performance of ConfSolve at scale using a virtual machine allocation problem. Such problems are commonplace in larger enterprises. ConfSolve performs well, finding solutions with up to 900 virtual machines allocated across 96 physical machines.

We compare ConfSolve with the closest existing work, Cauldron. This is in part to verify that ConfSolve can simply express such models. Benchmarking was also performed, with ConfSolve sustaining a 4x speed improvement.

Finally, the virtual machine allocation problem was benchmarked against Cauldron, in order to compare scaling performance between these tools. We found that Cauldron did not scale to problems of a practical size. ConfSolve is therefore the first general purpose configuration tool to do so.

Chapter 7

Extending ConfSolve for Reconfiguration

In this chapter an extension to the basic ConfSolve language which permits reconfiguration is described. We present a conceptual overview, example, syntax extensions, and a description of the translation of the extended language to MiniZinc.

7.1 Background

The configuration problems which we have examined so far have been one-off tasks; initial configurations of a new system, starting from scratch. In practice the majority of configuration tasks are incremental, starting from some existing state and applying changes which take the existing configuration into account.

Take for example, the virtual machine assignment problem from Section 6.2. After the system has been configured initially, it is desirable for subsequent *reconfigurations* to take into account the current allocation of virtual machines so as not to move virtual machines unnecessarily from one physical machine to the next. Such moves are to be expected, as the CSP solver may explore a different subsection of the solution space if we were to simply re-run a modified version of the initial configuration problem. Thus we must somehow inform the solver about the previous state of the system, and how it affects subsequent configuration decisions.

7.2 Extended ConfSolve

Given that re-configuration is fundamental to the configuration process, we have extended the basic ConfSolve language with variables and expressions which depend on the current state of the system:

parameters are variables which explicitly take their values from an external source

previous value expressions provide access to the prior ConfSolve solution

init and change blocks allow separate inital- and re-configuration constraints

Let us examine a simple example based on the virtual machine allocation problem, in which we allocate virtual machines across a rack of physical machines with a 4:1 ratio. Bin-packing constraints are ignored to keep the example minimal. The goal of this model is that a reconfiguration should never move a virtual machine off its physical host, unless that host machine has failed.

```
// model.csm
class Machine {
  param failed as bool;
}

abstract class VM {
  var host as ref Machine;
}

var rack1 as Machine[48];
var vms as VM[192];

change {
  forall vm in vms where !vm.host.failed {
    vm.host = ~vm.host;
  };
}
```

The `Machine` class represents physical machines, which may be in a `failed` state. We presume an external monitoring system determines this, and triggers a re-run of ConfSolve. The `VM` class represents a virtual machine, which has a physical `host` reference. Variables `rack1` and `vms` provide instances of both physical and virtual machines. Finally, the **`change`** block contains the constraints which are enforced when a re-configuration occurs (that is every configuration except the initial configuration).

The **change** constraint is a hard constraint specifying that if a virtual machine's host has not failed, then the value of `vm.host` must be equal to its value in the previous solution, $\sim$`vm.host`. The operator $\sim e$ yields the value of e in the previous solution to the model, may only be used within a **change** block, and assumes that the AST of the model has not changed: only parameters may change their values. Parameter values are described as CSON and provided in a separate file. The structure of the parameter file must match the structure of the ConfSolve model, such as the following extract:

```
// params.cson
{
  rack1: [
    Machine { failed: false },
    Machine { failed: true },
    Machine { failed: false },
    // ...
```

In which the second machine in `rack1` has failed.

7.3 Parameter changes and migrations

We now examine the concepts of extended ConfSolve formally: m is a ConfSolve model, P are the model's CSON parameters, and S is the model's previous CSON solution. In the initial instance S is undefined. We define two distinct kinds of re-configuration scenario:

parameter change is a reconfiguration triggered by a change in value of one of the model's parameters P. We presume some external system is monitoring and updating parameter values to reflect the current system state.

migration is a modification of the model m itself, altering its AST. The result is a new model. It is also possible to migrate the model's parameters P and previous solution S to correspond to the new AST where appropriate, for instance when renaming a variable.

These two re-configuration scenarios are illustrated in Figure 7.1. Both take into account the model m, previous solution S, and and parameters P. Thus when re-configuring the next ConfSolve solution S' is a function of m, S, and P.

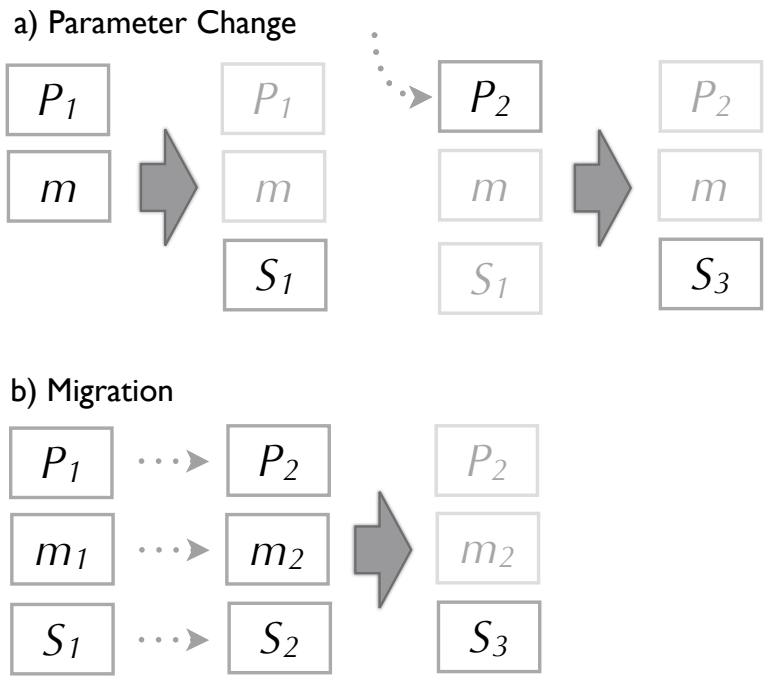

Figure 7.1: The two kinds of re-configuration. Bold arrows indicate ConfSolve compilation/solving. **a)** Initial configuration of model m with parameters P_1 results in solution S_1. Updated parameters P_2 are obtained, and a new solution S_3 found. **b)** Migration via modification of model m_1 to m_2 along with its parameters and solution is followed by re-solving to get solution S_3

By distinguishing parameter changes from migrations we separate re-configuration tasks into two kinds. The first being tasks which are explicitly captured in the model: the parameters which are expected to change regularly, perhaps every few minutes. The second being tasks which are less-common alterations of the model itself: migrations between different versions of the model. Examples of such migrations include changes to the infrastructure or the kinds of services present, *i.e.*, the removal and addition of new classes, objects and constraints.

7.4 An example translation

A simple example of a ConfSolve to MiniZinc translation is given below. A ConfSolve model contains four object instances of the class `Server` which has a solver-assigned id between 1 and 10. A reconfiguration constraint restricts the value of `id` to be equal to its previous value (if any):

```
class Server {
  var id as 1..10;
  change { id = ~id; }
}
var servers as Server[4];
```

The MiniZinc translation of this problem introduces the prior state in a variable prefixed with `_old` and translates the reconfiguration constraint accordingly. The individual fields are flattened into arrays, indexed by object. The object indices are in the range 1..4 as there are four instances of `Server` in the model:

```
// MiniZinc
1..4: servers = [1,2,3,4];
array[1..4] of var 1..10: Server_id;
array[1..4] of 1..10: old_Server_id = [2,4,6,8];

constraint
  forall (i in 1..4) (
    Server_id[i] = old_Server_id[i]
  );
```

7.5 Core Syntax extensions

This section presents the re-configuration extensions to the ConfSolve language formally, in the same manner as Section 5.1.

Additional Syntax of expressions:

$e ::=$	expression
$\sim e$	previous value

Extended ConfSolve introduces just one new expression, the *previous value* expression which evaluates to the value of some expression e in the previous solution to the ConfSolve model. It may only be used within a **`change`** block, introduced below.

Changes to the syntax of models:

Declaration ::=	declaration
ClassDecl	class decl.
EnumDecl	enum decl.
VarDecl	var decl.
ParamDecl	var decl.
Constraint	constraint
ConstraintBlock	constraint block
ClassDecl ::=	class decl.
abstract? **class** c **extends** c' {	
(VarDecl | ParamDecl | Constraint | ConstraintBlock)*	
}	
ParamDecl ::=	parameter decl.
param v **as** T	
param v **as ref** c	object reference
ConstraintBlock ::=	constraint block
change { Constraint* }	change block
init { Constraint* }	initial block

A number of changes to the syntax of models in Section 5.1 are made in extended ConfSolve. Parameters are introduced to complement variables; these take their values from an external CSON representation which follows the same structure as the current ConfSolve model. Constraint blocks are introduced at the global and class-nested levels, which provide conditional compilation of constraints. Constraints in a **`init`** block are enforced for the initial configuration only, while those in a **`change`** block are enforced only for subsequent re-configurations and so may contain the $\sim e$ operator to access the previous value of an expression.

7.6 Architecture

Our implementation of extended ConfSolve builds on top of the MiniZinc translation process and Gecode CSP solver described in Section 5.3. As shown in Figure 7.2 the basic process is extended by passing in both CSON-formatted parameters and the CSON-formatted previous solution into the ConfSolve compiler in addition to the ConfSolve model.

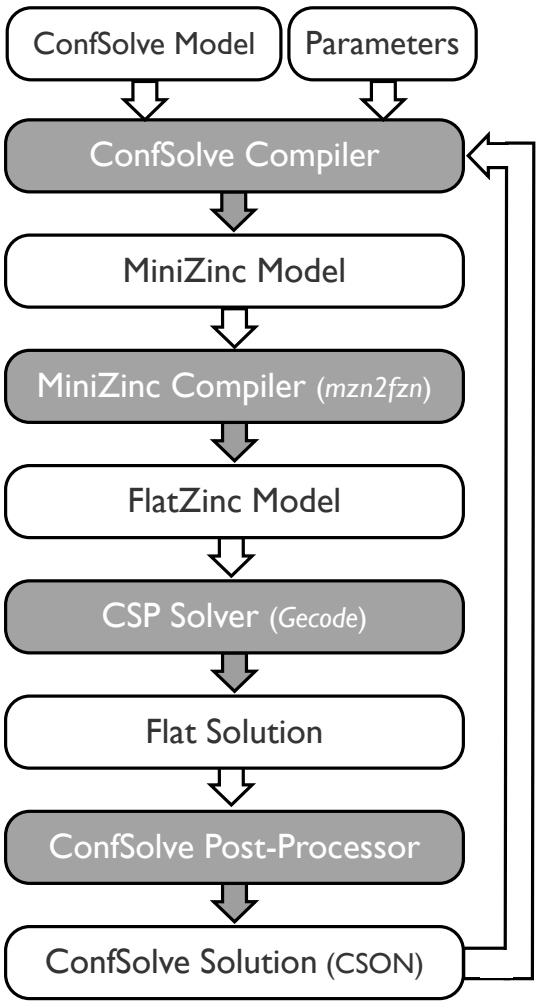

Figure 7.2: Compiling and solving an extended ConfSolve model. White boxes are files; shaded boxes are processes. The basic ConfSolve architecture has been extended so that both models (as ConfSolve) and parameters (as CSON) are input into the ConfSolve compiler, along with the previous solution (as CSON).

7.7 Translation to MiniZinc

As with ConfSolve, the semantics of extended ConfSolve are defined by specifying the translation of the language into MiniZinc. This sections consists of a number of extensions and modifications of the translation scheme given in section 5.4.5.

Syntax of CSON values:

V ::=	value
i	integer
true | **false**	boolean
$u.a$	enum member
c {Member*}	object
ref Target	object reference
$t[n]\{V_1,\ldots,V_n\}$	set literal
Member ::=	member
$v : V$	variable name : value
Target ::=	target
v	variable
Target.l	field access
Target$[i]$	set access

The process of translating extended ConfSolve to MiniZinc begins with not only a ConfSolve model, but also parameters and a previous solution, both expressed in CSON, the ConfSolve object notation. The syntax of CSON, defined in Section 5.4.6 is reproduced above for reference.

The translations outlined below occur after the static allocation phase described in section 5.4.4, in which indexes are assigned to each object, and the upper bound count(c) is calculated for each class c.

Translation of CSON values $[\![V]\!]$:

Given a CSON value V, its translation to MiniZinc $[\![V]\!]$ is defined as:

when $V =$ **true** $\vee$ $V =$ **false** $\vee$ $V = i$

V

when $V = c$ {Member*} $\vee$ $V = c[n]\{V_1,\ldots,V_n\}$

undefined

when $V = t[n]\{V_1,\ldots,V_n\}$

{ for $V_i \in \{V_1,\ldots,V_n\}$, $[\![V_i]\!]$ **}**

when $V = u.a$

$eindex(u, a)$

when $V =$ **ref** Target

the index of the object O where $path(O)$ = Target

We first define the MiniZinc translation $[\![V]\!]$ of a ConfSolve value V, which is used throughout the following translations. Boolean and integer values are translated as literals. The translation of objects and sets of objects is undefined. Firstly because their MiniZinc representation consists of a constant integer index (or a set of these), which is already known from the ConfSolve model's static allocation phase. Secondly, the expansion of the object's fields into arrays is handled later as part of the translation of variables, so there is not a one to one correspondence between CSON values and MiniZinc values in this case.

The translation of enumerations is the same as for the ConfSolve model: $eindex(u, a)$ is the index of element a in the declaration **enum** u $\{a_1 \ldots a_n\}$, as in section 5.4.5.

The translation of an object reference is the index of the object in the ConfSolve model whose path matches the Target expression which may be either a variable, a field access, or an indexed object-set access. Formally, we define it as the object O with $path(O) =$ Target where $path(O)$ is the CSON Target expression corresponding to the full path of the object within the global scope. Each object has a unique Target path, for example `rack[2].machines[4].hostname`.

7.7.1 Parameters

The values of all ConfSolve parameters are contained in a single CSON AST, which follows the same structure as the original model. The type of all parameter values must match their declared type, *i.e.*, **param** v **as** T. Parameter values do not affect the object counting performed in the static allocation phase, because that is based on the type only, and not the value. Sets of objects already have a fixed cardinality as part of their type, so there is no way to introduce extra objects.

The translation of parameters is described as two new rules, which handle **param** v **as** T, in both class-level and global scopes. These rules make use of the *corresponding CSON value* for a given parameter, that is, the CSON value which has the equivalent path in the CSON AST as the given parameter. If such a CSON value does not exist for every parameter, then the translation cannot proceed.

Translation of global parameter declarations:

For each global declaration **param** v **as** T, and corresponding CSON value V, introduce a declaration:
when $T = c[n] \vee T = c$
the translation of global **var** v **as** T, according to Section 5.4.5
otherwise
$[\![T]\!] : v = [\![V]\!]$

For each global declaration **param** v **as ref** c, and corresponding CSON value V, introduce a declaration:
$[\![c]\!] : v = [\![V]\!]$

Objects and sets of objects are translated in the same manner as **var** declarations in standard ConfSolve (see Section 5.4.5). This is due to the fact that their translation consists of constant object indexes generated in the static allocation phase, rather than user-specified CSON values such as integers, sets of integers, and booleans. The purpose of an object parameter is therefore to allow the values of its fields to be specified as CSON.

For all other types, parameters are translated into MiniZinc constants, which consist of a compound declaration and assignment. Reference parameters are translated in a similar manner, in which the CSON path to the object is mapped into an object index via $[\![V]\!]$.

Translation of class-level parameter declarations:

For each ClassDecl defining a class c where $count(c) > 0$, containing fields **param** f_i **as** ${T_i}^{i \in 1..n}$, and corresponding CSON values ${V_j}^{j \in 1..n}$, introduce a declaration for each field f_i:
when $T = c[n] \vee T = c$
the translation of class-level **var** v **as** T, according to Section 5.4.5.
otherwise
array [1 **..** $count(c)$**] of** $[\![T_i]\!] : c\_f_i =$ **[** for $j \in 1..count(c), [\![V_j]\!]$ **]**

Where class c contains fields **param** f_i **as ref** ${c_i}^{i \in 1..n}$, introduce a declaration for each field f_i:
array [1 **..** $count(c)$**] of** $[\![c_i]\!] : c\_f_i =$ (as above)

Variables nested within classes are translated using an **array** containing all instances of a particular field, where the object index is the index into the array. Objects and sets of objects are once again translated in the same manner as **var** declarations in standard MiniZinc (Section 5.4.5). All other types are translated as an **array** for each field, containing the translated CSON value for each object index in $1..count(c)$. Reference parameters are translated in the same manner.

7.7.2 Previous values and change expressions

The previous solution to a ConfSolve model is represented as a CSON AST. This is the output of a previous run of ConfSolve. As with parameters, the values in the CSON solution does not affect the object counting performed in the static allocation phase, because it is based on type only, and not on value. The number of objects is therefore fixed.

We assume that the AST of the model has not changed since it was used to generate the previous solution. However, the value of the parameters may change freely. We also include some relaxations of this assumption which allow for easier *migrations*.

The translation of previous value expressions, and change blocks is described by both new rules and revisions of existing rules for translating variables and expressions.

Translation of change and init statements:

Iff performing a reconfiguration, for each statement **change** { $e_i \in$ Constraint* }, introduce a expression:

$e_1 \wedge \cdots \wedge e_n$

Iff performing an initial configuration, for each statement **init** { $e_i \in$ Constraint* }, introduce a expression:

$e_1 \wedge \cdots \wedge e_n$

Change and init statements are blocks containing constraints, and can appear both at the global level and nested within classes. Their translation is a simple form of conditional compilation: change blocks are translated as the conjunction of their constraint expressions only when a re-configuration is being performed, otherwise they are not translated. Likewise for init blocks.

Additional translation of global variable declarations:

For each global declaration **var** v **as** T, where $T \neq c[n] \wedge T \neq c$, and corresponding CSON value V, introduce a declaration:

$[\![T]\!]$: `old_`$v = [\![V]\!]$

For each global declaration **var** v **as ref** c, introduce a declaration:

$[\![c]\!]$: `old_`$v = [\![V]\!]$

A new translation step is defined for global variables, in addition to those in standard ConfSolve (Section 5.4.5). For global variables which are not objects or sets of objects, a MiniZinc constant is introduced with the appropriate type, and with value equal to the MiniZinc translation of its corresponding CSON value. The MiniZinc variable name is prefixed with `old_` as its purpose is to expose the previous value of that variable within the MiniZinc model. Reference variables undergo an equivalent translation.

Additional translation of class-level variable declarations:

For each ClassDecl defining a class c where $count(c) > 0$, containing fields **var** f_i **as** ${T_i}^{i \in 1..n}$ and corresponding CSON values ${V_j}^{j \in 1..n}$, where $T_i \neq c[n] \wedge T_i \neq c$, introduce a declaration for each field f_i:

array [1 **..** $count(c)$**] of** $[\![T_i]\!]$: `old_`c`_`f_i = **[** for $j \in 1..count(c), [\![V_j]\!]$ **]**

If any corresponding CSON value V_j is undefined, then substitute the anonymous variable _ in place of $[\![V_j]\!]$ and introduce the constraint:

constraint c_f_i **[** j **]** = `old_`c`_`f_i **[** j **]**

Where class c contains fields **var** f_i **as ref** ${c_i}^{i \in 1..n}$, introduce a declaration for each field f_i:

array [1 **..** $count(c)$**] of** $[\![c_i]\!]$: `old_`c`_`f_i = (as above)

An equivalent translation step is introduced for class-level variables, resulting in the introduction of `old_` prefixed MiniZinc variables. ConfSolve variables nested within classes are translated using an **array** containing all instances of a particular field, where the object index is the index into the array. Variables which declare objects or sets of objects do not have a translation, because their translation consists of constant object indexes generated in the static allocation phase, as was the case with the translation of parameters.

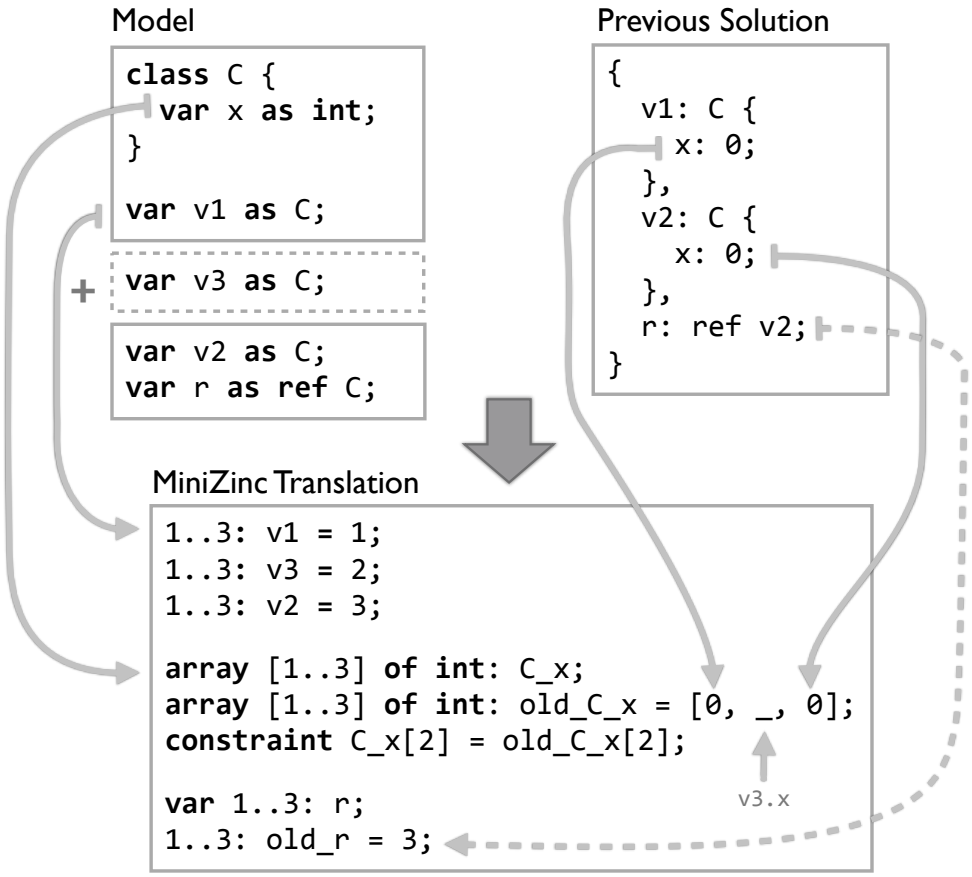

Figure 7.3: The translation of previous values of a class-level variable x, as part of a migration in which a new object v3 is added to the model.

There is a special provision made for when the CSON value V_i corresponding to field f_i is not present in the previous solution's CSON AST. This is to support migrations in which variables declaring objects are added and is the only situation in which this can occur. See Figure 7.3 for an illustration of a migration in which an new object `v3` is added to an existing model for which a previous solution already exists. In order to translate the declaration of `x`, we first attempt to find the corresponding CSON value in the previous solution by traversing its AST. For `v1.x` and `v2.x` this is successful, however for `v3.x` it is not. Instead we use MiniZinc's anonymous variable `_` as the translation of `v3.x` and introduce a constraint that the new and old values of `v3.x` are equivalent. This is a symmetry breaking constraint which prevents the solver from searching for satisfying old values of `v3.x`. The benefit of this approach over assigning an arbitrary placeholder value for missing values is that the count of differences between the `old_` variables and their counterparts is equal to the number of changes between the previous solution and the forthcoming solution, which we take advantage of later.

Translation of expressions $[\![e]\!]$:

$[\![v]\!] \triangleq$ (see below)

$$[\![e.l]\!] \triangleq \begin{cases} \texttt{old\_}classof(e)\texttt{\_}l\,\mathbf{[}[\![e]\!]\mathbf{]} & \text{if } e.l \text{ is a sub-expression of some } {\sim}e' \\ classof(e)\texttt{\_}l\,\mathbf{[}[\![e]\!]\mathbf{]} & \text{otherwise} \end{cases}$$

$[\![{\sim}e]\!] \triangleq$ (see below)

We define a revised translation of variables $[\![v]\!]$ below. The previous value expression $\sim e$ is defined as simply $[\![e]\!]$, but the rules for both variable (v) translation and member expression $e.l$ translation are made context-dependent to make use of previous values. In the translation of member expressions, there are now two cases. The second case is the standard translation which is an array access where $[\![e]\!]$ is the object index. The first case, however, applies whenever the member expression is a sub-expression of a previous-value expression, and results in the same translation except that the resulting MiniZinc variable receives the `old_` prefix. The mechanism for accessing previous values can now be seen: it is enough to prefix any variable with `old_` to access its prior value which was introduced during the translation of variable declarations.

Translation of variables $[\![v]\!]$:

Within the scope of class c, the translation $[\![v]\!]$ of a variable v is:
when v is a sub-expression of some $\sim e$, and v is not quantified:
  when v is declared in class $c' \in c^*$
    `old_`c'`_`v[this]
  otherwise
    `old_`v
otherwise
  when v is declared in class $c' \in c^*$
    c'`_`v[this]
  otherwise
    v
Where c^* is the set containing c and all its ancestors.

The translation of variables undergoes a similar modification as that of member expressions. When a variable is a sub-expression of a previous-value expression $\sim e$ then its standard translation is prefixed with `old_`. The standard translation has two cases; the first for class-level variables, and the second for global variables.

We impose an additional restriction on when variables within a previous-value sub-expression are translated with the `old_` prefix. If a variable is quantified, *i.e.* was introduced by a Fold expression, such as x in the expression **forall** $(x$ **in** $e_1)$ (e_2), then it is not translated in this manner. This is because quantified variables are bound to the expression over which they quantify. Thus it is not meaningful to talk about the previous value of a quantified variable such as $\sim x$, instead we must quantify over $\sim e_1$. This approach has the benefit of not adding any further complexity to the translation of folds.

7.8 The min-changes heuristic

There is one final translation which we have not yet mentioned, the *min-changes* heuristic. This gives us the ability to automatically infer **`change`** constraints for a model, without the user having to specify them. The min-changes heuristic is simple: for each variable in the model a soft constraint is introduced which is satisfied if the value of the variable has not changed from the previous solution. Unlike the

other translation steps the min-changes heuristic is implemented as a ConfSolve-to-ConfSolve transformation; we simply expand variable declarations into variable declarations plus constraints, before continuing with the usual MiniZinc translation.

Expansion of global variable declarations:

For each global declaration **var** v **as** T where $T = c[n] \vee T = c$ and each global declaration **var** v **as ref** c, introduce a global constraint:

change {
  maximize bool2int$(v = \sim v)$
}

For global variables a constraint is introduced which maximises the integer equivalent of an equality constraint between the value of the variable and its previous value. If the variable has not changed, then the expression evaluates to 1, otherwise 0. The constraint is placed within a **`change`** block as it makes use of the $\sim$ operator. Constraints are not placed over objects or sets of objects as these represent locations, not values.

Expansion of class-level variable declarations:

For each ClassDecl defining a class c where $count(c) > 0$, containing fields **var** f_i **as** ${T_i}^{i \in 1..n}$ where $T_i \neq c[n] \wedge T_i \neq c$, and fields **var** f_j **as ref** ${c_j}^{j \in 1..m}$, introduce a class-level constraint for each field f_x:

change {
  maximize bool2int$(f_x = \sim f_x)$
}

The same process is repeated for class-level variables (fields), where a constraint is introduced for each.

The min-changes heuristic provides a simple way to automatically generate change constraints for models. It acts to resist change across all variables, and so can be expected to be useful only in cases where change is universally undesirable. We examine the suitability and performance of this heuristic in the next chapter.

Chapter 8

Evaluation of Reconfiguration with ConfSolve

In this chapter the reconfiguration extensions for ConfSolve are evaluated against several system configuration problems. The problems from Chapter 6 are expanded to cover reconfiguration and the three scenarios of parameter change, migration, and a combination of the two.

8.1 Reconfiguration strategies

In order to evaluate our implementation of extended ConfSolve we examine both the quality of solutions and the impact of reconfiguration on performance. Solution quality is measured by counting the number of unnecessary changes introduced by a reconfiguration. We compare three reconfiguration strategies:

none ignores the previous system state

custom uses a model's custom `change` expressions, to access previous system state

automatic uses a simple heuristic in place of a model's `change` expressions

The *automatic* heuristic introduces a **change** constraint for each variable, given by the following ConfSolve declaration:

$$\textbf{maximize bool2int}(v = \sim v).$$

This allows us to measure the effectiveness of having custom **change** constraints as a language feature, rather than having them determined by the compiler automatically.

We examine three reconfiguration scenarios, which represent the three modes in which ConfSolve can operate: a migration, a parameterised model, and a migration of a parameterised model.

8.2 Experimental setup

The evaluation was performed on a machine with a 2.5GHz Intel Core 2 Quad processor and 8GB of RAM, running Ubuntu 12.04. We used the 64-bit MiniZinc to FlatZinc converter version 1.6.0 with the `--no-optimize` flag, and the 64-bit Gecode FlatZinc interpreter version 3.7.3.

8.3 Adding Virtual Machines

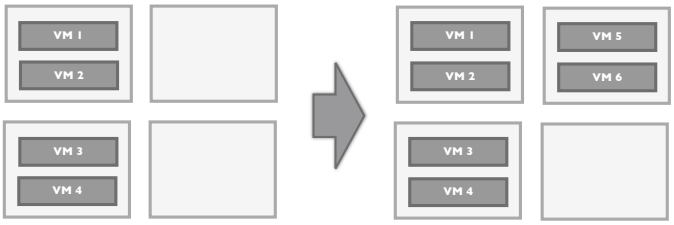

Figure 8.1: Two virtual machines are added to an existing half-full datacenter.

This evaluation extends an evaluation used in the original ConfSolve paper [Hewson, Anderson, and Gordon, 2012] to include reconfiguration via a migration. We use ConfSolve to generate an assignment of virtual machines to physical machines in an Infrastructure as a Service (IaaS) configuration, and then add more virtual machines as a migration. Figure 8.1 illustrates this scenario.

Each physical machine is identical, having 8 CPUs and 16GB or memory. Each virtual machine has variables representing its requirements on the physical machine resources. These are declared as follows:

```
class Machine {
  var cpu as int;          // 1/2 cores
  var memory as int;       // MB
  var disk as int ;        // GB

  cpu = 16;                // 2x Quad Core (1/2 core units)
  memory = 16384;          // 16 GB = 4x4GB DIMM
```

```
  disk = 2048;            // 2 TB
}

abstract class VM {
  var host as ref Machine;
  var disk as int;
  var cpu as int;
  var memory as int;
}

class StandardVM extends VM {
  cpu = 1;
  memory = 768;
  disk = 20;
}
```

The infrastructure consists of two racks of 48 physical machines, onto which we wish to allocate 250 virtual machines:

```
var rack1 as Machine[48];
var rack2 as Machine[48];

var vms as StandardVM[250];
```

A bin-packing constraint on virtual machine placement prevents over-provisioning of host resources, *i.e.*, for each physical machine the sum of required resources must be less than the amount provided by the machine:

```
var machines as ref Machine[48];

forall m in machines {
  sum r in vms where r.host = m {
    r.cpu;
  } <= m.cpu
  &&
  sum r in vms where r.host = m {
    r.memory;
  } <= m.memory
  &&
  sum r in vms where r.host = m {
    r.disk;
  } <= m.disk;
};
```

Finally, a reconfiguration constraint, stating that each virtual machine should remain on its previous host:

```
change {
  forall vm in vms {
    vm.host = ~vm.host;
  };
}
```

To perform the evaluation, the size of the virtual machine set `VM[250]` is incrementally increased by editing the model file. The results of scaling this problem up to 500 virtual machines are shown in Figure 8.2. The custom **change** expressions outperform both the *automatic* and *none* strategies, with regard to both time and memory. The automatic approach quickly reaches the solver timeout of 10 seconds before completing its search, though it is still able to produce sub-optimal results as it progresses.

The quality of the solutions follows a similar trend, with the the custom **change** expressions performing a perfect reconfiguration with no reassignments of existing machines. The *automatic* strategy quickly tends towards 250 reassignments, the maximum possible. The *none* strategy levels-off at 50 reassignments, which reflects the default behaviour of the Gecode solver.

The poor performance of the *automatic* strategy is due to extraneous constraints. The strategy generates a constraint for each variable in the model, yet most of these variables are in fact constants, such as `cpu = 1`. This results in an ineffective branch-and-bound search by the solver, which allocates a large amount of memory for backtracking. Ultimately the search times-out before finding high quality solutions. However, this behaviour is problem-specific and due to the internal heuristics of the solver. Redundant constraints do not in general necessarily have a negative impact on performance.

We conclude that for this evaluation, the addition of custom **change** expressions to ConfSolve results in both an improvement in time and, in particular, memory performance of the solver and successfully prevents unnecessary configuration changes when compared with the *none* and *automatic* strategies.

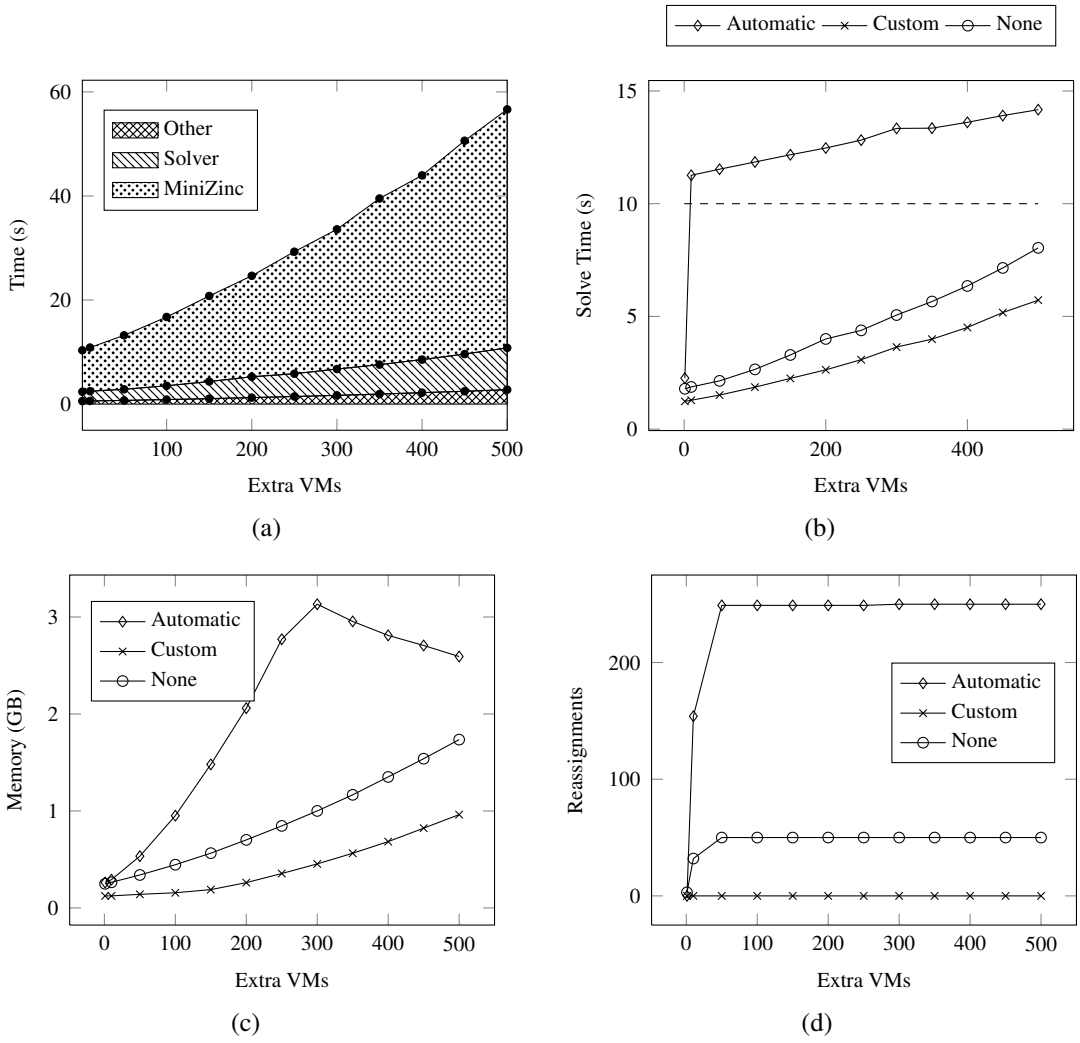

Figure 8.2: Adding Virtual Machines. **a)** Total time, including the time for MiniZinc compilation, which dominates the process. **b)** Solve time, with time-out at 10 seconds indicated by the dashed line. The *automatic* strategy quickly times-out. **c)** Solver memory usage, with high memory usage for the *automatic* strategy. The unusual shape of this graph is due to some internal aspect of the Gecode solver. **d)** Solution quality, with 250 reassignments for the automatic strategy, and 50 for *none*.

8.4 Parameters: Virtual Server Failure

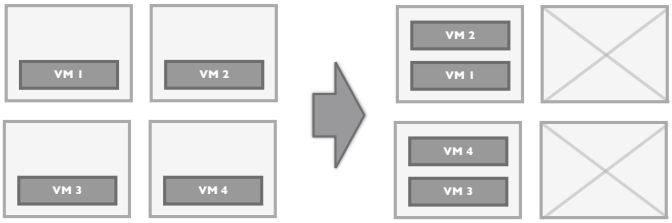

Figure 8.3: Two physical machines fail, causing a reassignment of virtual machines.

This evaluation extends an evaluation used in the original ConfSolve paper [Hewson, Anderson, and Gordon, 2012] to include reconfiguration via parameterisation. We use ConfSolve to generate an assignment of virtual machines to physical machines in an Infrastructure as a Service (IaaS) configuration, and then fail some of the machines via a parameter. Figure 8.3 illustrates this scenario.

Each physical machine is identical, having 2 CPUs and 2GB or memory. A parameter `online` indicates whether a machine is online or has failed:

```
class VM {
  var host as ref Machine;

  var cpu as int;
  var memory as int;
  var disk as int;

  cpu = 1;
  memory = 1024;
  disk = 5;
}
```

Each virtual machine is identical and has variables representing its requirements on the physical machine resources:

```
class Machine {
  param online as bool;

  var cpu as int;
  var memory as int;
  var disk as int;

  cpu = 2;
```

```
  memory = 2048;
  disk = 10;
}
```

The infrastructure consists of 200 physical machines, onto which we wish to allocate 200 virtual machines, with a 2:1 ratio this means that the physical machines are at 50% capacity:

```
var machines as Machine[200];
var vms as VM[200];
```

A bin packing constraint identical to that in section 8.3 is added to the model, which we do not show here. This ensure that physical machines are not over-provisioned.

Virtual machines are constrained to be hosted only on machines which are online:

```
forall vm in vms {
  vm.host.online = true;
};
```

We wish to distribute the virtual machines across the infrastructure, leaving headroom, rather than packing them tightly on to physical machines. As this is a preference we make use of a soft constraint, to minimise the number of virtual machines with a common host:

```
minimize sum vm1 in vms {
  count (vm2 in vms where vm1.host = vm2.host);
};
```

Finally, a reconfiguration constraint, which requires each virtual machine to remain on its previous host as long as that host was previously online and is so currently. The intent of this constraint is the same as for the previous example, the additional complexity comes from the need to take into account machine failure:

```
change {
  forall m in machines where m.online && ~m.online {
    forall vm in vms where ~vm.host = m {
      vm.host = ~vm.host;
    };
  };
}
```

To perform the evaluation, the sizes of both the set of physical machines and the set of virtual machines are simultaneously increased in size, while maintaining their ratios. An initial configuration is performed, followed by a reconfiguration in which 50% of the machines have their `online` parameter set to false.

The results of scaling this problem up to 300 virtual machines are shown in Figure 8.4. Custom **change** expressions narrowly outperform the *automatic* and *none* strategies with regard to time, diverge towards significant improvements with regard to memory from around 150 machines.

The quality of the solutions forms two distinct categories. The custom **change** expressions perform a perfect reconfiguration, in which only 50% of the virtual machines are reassigned. The *automatic* and *none* strategies both perform the maximum possible number of reassignments, 100%.

We conclude that for this evaluation, custom **change** expressions are a valuable addition to ConfSolve, resulting in a minimal reconfiguration, where a maximal one would otherwise have occurred.

8.5 Migration with Parameters: Cloudbursting

This evaluation combines a migration with parameter changes, in order to show that both can occur simultaneously. We model a scenario known as *cloudbursting* in which excess load from an enterprise datacenter may be run on the cloud. Figure 8.5 illustrates this scenario.

We create an abstract class to represent a host, which may be either a physical machine with 4 CPUs and 4GB of RAM, or a cloud, which has no fixed resources. Physical machines have an `online` parameter:

```
abstract class Host {}

class Machine extends Host {
  param online as bool;
  var cpu as int = 4;
  var memory as int = 4096;
}

class Cloud extends Host {}
```

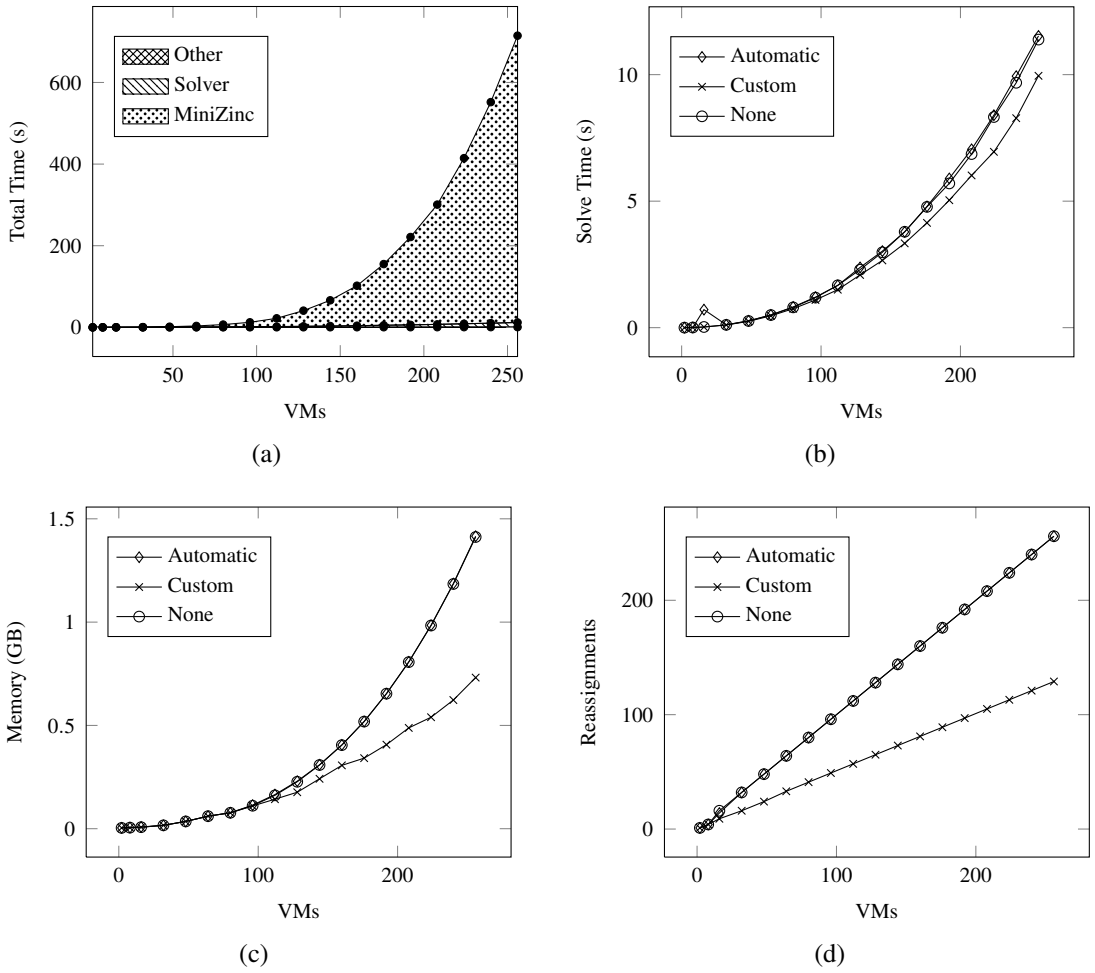

Figure 8.4: Virtual Server Failure. **a)** Total time, including the time for MiniZinc compilation, which dominates the process. **b)** Solve time, showing a marginal advantage to the *custom* strategy. **c)** Solver memory usage, with a significant advantage to the *custom* strategy. **d)** Solution quality, showing *custom* achiving 50% reassignments, the minimum possible.

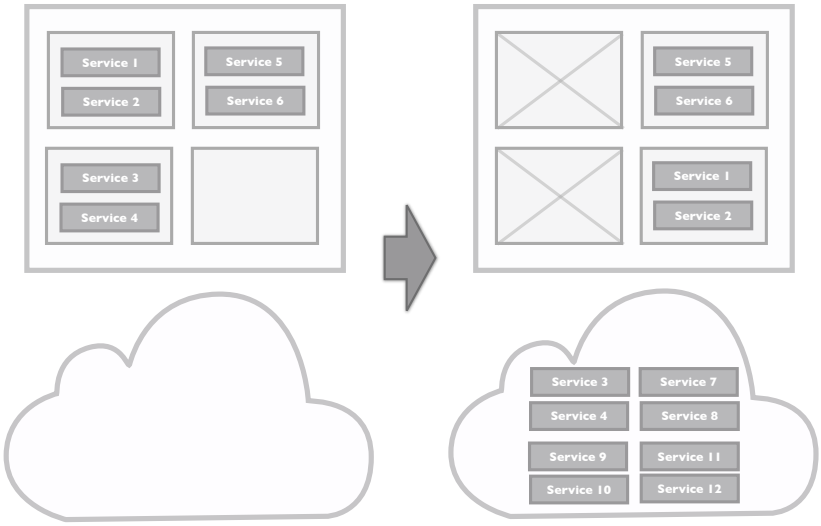

Figure 8.5: Two enterprise machines fail, at the same time as more services are added. Services on failed machines are moved onto non-failed machines, or into the cloud.

Services are tasks which can be placed on hosts, and have requirements on the amount of CPU and memory required to run them:

```
abstract class Service {
  var host as ref Host;
  var cpu as int;
  var memory as int;
}
```

We define three specific types of task: web, worker, and database, with differing CPU and memory requirements:

```
class Web extends Service {
  cpu = 2;
  memory = 2048;
}

class Worker extends Service {
  cpu = 2;
  memory = 2048;
}

class Database extends Service {
  cpu = 4;
  memory = 4096;
}
```

The infrastructure consists of 300 physical machines within the enterprise, and a single cloud provider:

```
var enterprise as Machine[300];
var cloud as Cloud;
```

We create web, worker, and database services in the ratio 2:2:1.

```
var webs as Web[200];
var workers as Worker[200];
var databases as Database[100];
```

So that we may more easily quantify over all services, a set of `service` references is created, which the solver will automatically resolve to the declarations above:

```
var services as ref Service[500];
```

A bin-packing constraint for the services hosted in the enterprise should be familiar from the previous examples:

```
forall m in enterprise {
  sum s in services where s.host = m {
    s.cpu;
  } <= m.cpu
  &&
  sum s in services where s.host = m {
    s.memory;
  } <= m.memory;
};
```

We do not wish to host services in the cloud if there is available capacity within the enterprise. The constraint below states that if the number of services hosted in the cloud is greater than zero, then the number of services hosted in the enterprise is equal to the number of enterprise machines which are online. ConfSolve uses the `->` operator for logical implication:

```
count (s in services where s.host = cloud) > 0 ->
  count (s in services where s.host in enterprise) =
    count (m in enterprise where m.online);
```

We constrain services to be placed only on machines which are online:

```
forall s in services {
  s.host.online = true;
};
```

Finally, a reconfiguration constraint, similar to that from section 8.4, which requires each service to remain on its previous host as long as that host was previously online and is so currently. This applies only to machines within the enterprise, not the cloud:

```
change {
  forall m in enterprise where m.online && ~m.online {
    forall s in services where ~s.host = m {
      vm.host = ~s.host;
    };
  };
}
```

To perform the evaluation, an initial configuration is performed, after which the number of `Worker` services is doubled by manually editing the model, as a migration. Additionally, 50% of the machines have their `online` parameter set to false.

The results of scaling the problem up to 300 machines are shown in Figure 8.6. With regard to time, custom **change** expressions narrowly outperform the *none* strategy, while the *automatic* strategy tends rapidly towards a solver timeout at 60 seconds. In terms of memory performance, custom **change** expressions significantly outperform both of the other strategies, showing much better scaling.

The quality of the solutions follow a new pattern. Both the *custom* and *automatic* strategies achieve a perfect reconfiguration, in which only 50% of the services are reassigned. However, the automatic strategy yields worse results after it starts to timeout at 250 machines. The *none* strategy remains poor, performing 100% reassignments.

As before, the performance of the *automatic* strategy is due to extraneous constraints. Most variables are in fact constants, such as `cpu = 2`, yet a constraint is generated to optimise towards keeping the value from changing. The resulting search performed by the solver is ineffective, resulting in long solve times and eventually timeout. However, the behaviour of the solver in such cases is problem specific, extraneous constraints not necessarily have a negative impact on performance in general.

We conclude that for this evaluation, custom **change** expressions show their value in terms of performance, even though the *automatic* strategy is able to provide results

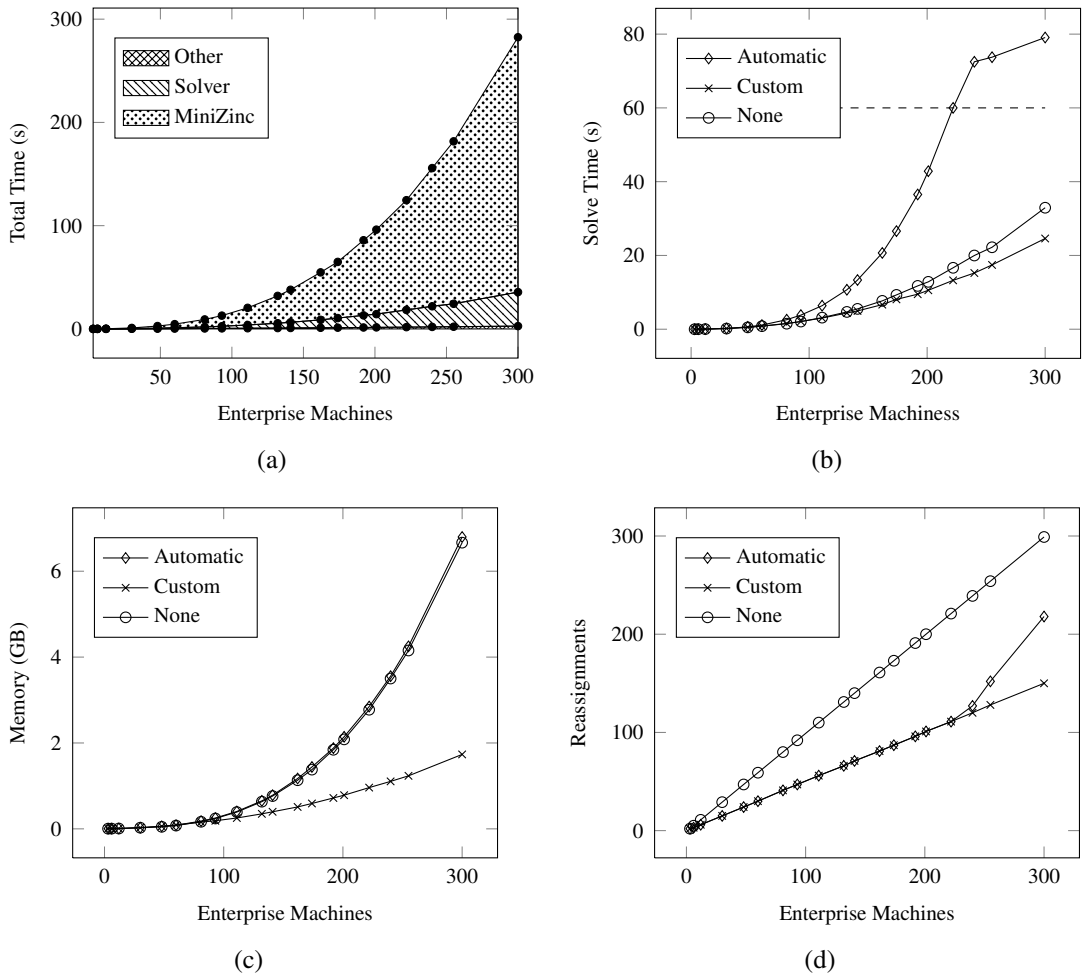

Figure 8.6: Cloudbursting. **a)** Total time, including the time for MiniZinc compilation, which dominates the process. **b)** Solve time, showing the *automatic* strategy timing-out at 60 seconds. **c)** Solver memory usage, with a significant advantage to the *custom* strategy. **d)** Solution quality, showing *custom* and *automatic* achieving 50% reassignments, the minimum possible, until *automatic* times-out at around 225 machines.

of the same quality. In practice, this problem is most likely to be memory-bound, thus the *custom* strategy offers desirable benefits.

8.6 Summary

The purpose of this evaluation has been to quantify the benefit of incorporating reconfiguration into the ConfSolve language. We have compared a ConfSolve model which takes advantage of a custom reconfiguration goal with two simpler alternatives: a "from-scratch" reconfiguration which ignores prior state, and a naive heuristic which

takes into account prior state without user interaction.

We compare these strategies across a several virtual machine allocation tasks, which are extensions of the problem used in Section 6.2 to evaluate the basic ConfSolve language. These extensions represent the three reconfiguration scenarios applicable to extended ConfSolve: migration, parameter change, and a combination of the two.

We found that making use of ConfSolve's custom reconfiguration primitives resulted in better performance in terms of memory, time, and solution quality, depending on the task. Though there may well be examples where this is not the case.

Chapter 9

Conclusions

This thesis provides an approach to using constraint-based methods to automate system configuration tasks. We can confirm that constraint programming is indeed a viable approach to large-scale configuration management, at least at the scale seen in the average enterprise. We have shown that complex constraints can be integrated into a familiar configuration language without exposing the system administrator to the underling complexity of constraint programming. Finally, we provide a novel approach to capturing state within a configuration language, so that automated configuration is not just a one-off task, but can be performed repeatedly, with minimal impact. This was perhaps the last significant barrier to the adoption of such tools.

The main contributions are divided into two sections, a division corresponding to the existing literature and the limitations of previous work. Firstly, we define a configuration language, ConfSolve, in which declarative configuration tasks are expressed as an object-oriented model with logical, arithmetic, and set theoretic constraints. Furthermore, soft constraints may be described in terms of maximisation or minimisation of constraint expressions. We then evaluate this language against existing configuration problems from the literature, and against larger virtual infrastructure problems. Secondly, we extend the ConfSolve language to reconfiguration tasks, in which the current system state is taken into account. This is achieved by modelling reconfiguration as a constraint optimisation problem and exposing language primitives to the user in order to allow him to customise the decision-making process. We evaluate the extended language features against a set of configuration problems and find that the user-customised reconfiguration process outperforms a naive "minimise all changes" heuristic, as well as a from-scratch reconfiguration.

9.1 Answers to research questions

Each of the research questions from Chapter 1 has been answered somewhere in this thesis. Explicit answers to the questions are given below, as a summary of the contributions made in this thesis:

1. *Identify where constraint-based approaches to system configuration may be advantageous.*

   Virtualised infrastructure is a natural source of configuration problems which may be automated. This is in part because software infrastructure is easier to automate than its hardware equivalent, but also because virtual machines act as an atomic unit of configuration, to be located wherever is convenient. Furthermore, virtual infrastructure is not a one-off task, lends itself well to reconfiguration problems both in terms of scaling up and responding to failure.

2. *Investigate the suitability of existing constraint-based techniques for solving configuration problems.*

   We have found constraint programming to be a capable approach to solving configuration problems. The Gecode constraint solver performs well with problems of up to 1000 machines. The MiniZinc language, although it has scope for performance improvement, is expressive and convenient as an intermediate language for modeling constraint programming problems.

3. *Propose a method which can apply the identified constraint-based techniques to the identified configuration problems.*

   We define a constraint-based object-oriented configuration language, ConfSolve and a translation of the language to a constraint satisfaction problem encoded in MiniZinc. This serves as an executable semantics of the ConfSolve language, and provides a level of formalism beyond that published in previous work. Furthermore, we define an extension to the ConfSolve language which incorporates reconfiguration via state-aware constraints.

4. *Design and implement a demonstration tool based on the proposed methods.*

   We developed the ConfSolve compiler which translates a high-level object oriented specification expressed in the ConfSolve language into a lower-level constraint satisfaction problem expressed in MiniZinc. The ConfSolve compiler is

implemented in approximately 2000 lines of OCaml, and its functional implementation closely represents the formalised description of the translation process.

5. *Evaluate the tool against the identified configuration problems.*

   Evaluation of ConfSolve is performed against several virtual infrastructure problems, and a suite of problems used in previous work. We show that ConfSolve models can scale to problems of a useful size, and that can model problems from previous work with high performance. ConfSolve reconfiguration primitives can offer a performance benefit over a from-scratch configuration and a naive reconfiguration heuristic.

This thesis also tackles the open questions from Chapter 1, with the following contributions:

6. *Soft constraints, which need not be fully satisfied, need to be able to be modelled and solved.*

   ConfSolve models configuration tasks as a constraint optimisation problem which naturally handles soft constraints as maximisation or minimisation of an objective function. This capability goes beyond previous work and is fundamental to support for reconfiguration.

7. *Small changes to the configuration problem should result in small changes to the resulting solution.*

   We define an extension to the ConfSolve language which incorporates reconfiguration via state-aware constraints. A translation process is described for these extensions, which encodes the reconfiguration process as a constraint optimisation problem. The optimisation goal is to produce a new configuration as close as possible to the current system state. The user may customise this goal using novel language primitives.

8. *The configuration interface needs to be appropriate for the intended users. Thus novel features need to introduce a minimal amount of complexity.*

   The ConfSolve language follows an object-oriented approach seen in existing configuration languages. Representing soft constraints as minimisation and maximisation is natural, and should be familiar to system administrators who have

a knowledge of database query languages. ConfSolve's reconfiguration primitives introduce state into the configuration process in a minimal and customisable manner.

9.2 Further Work

There are several aspects of this research which lead to new questions and potential for future work. The first is to investigate the performance of ConfSolve using constraint solvers other than Gecode. Gecode has an excellent reputation and we found no other solvers with fully compliant FlatZinc implementations which were even within an order of magnitude of Gecode's performance. However, solvers are evolving constantly, and so there is scope for benchmarking against other solvers.

The performance of the MiniZinc compiler leaves much to be desired, and we see no fundamental reason why it cannot scale significantly better, though this is a task for that compiler's authors. Another direction is to modify the translation process so that ConfSolve targets another constraint language beyond MiniZinc. Indeed we have recently collaborated with an MSc student who has hand-written a number of virtual machine configuration problems using Answer Set Programming (ASP) in order to explore the approach's expressiveness and performance [Xu, 2012]. The results were positive. There are yet other approaches to constraint solving: an interesting case is very large scale problems, for which a complete search is infeasible. Such problems can be solved by local search algorithms, such as simulated annealing. Local search is typically specified in an imperative manner, specifying the specifics of the search strategy. There is an opportunity for research into creating a declarative interface to local search solvers, with some promising research already underway [Benoist, Estellon, Gardi, Megel, and Nouioua, 2011].

The ConfSolve compiler does not perform any optimisations beyond the representation it uses for flattened object-oriented models. One powerful technique available in constraint programming is the global constraint. A global constraint involves several variables, the classic example being the *all_different* constraint. Global constraints are implemented by custom propagators inside the solver, which result in more efficient search. ConfSolve could provide higher-level operators which map to such constraints. Alternatively, the compiler could attempt to detect that a given expression is a decomposition of a given global constraint, though the complexity of such an analysis may well be prohibitive.

A practical concern regarding the use of constraint-based configuration in a real-world setting is that constraint solvers return either a solution or the message *unsatisfiable*. The process of debugging an incorrect model is therefore somewhat difficult. The user must remember which parts of the model they changed since the last solution attempt, and deduce what is wrong with them. SAT solvers, however, are able to extract *minimal unsatisfiable cores* which identify problematic terms. Similar methods now exist for CSP, but have yet to be implemented by a popular solver [Shah, 2011].

One relatively straightforward feature which is missing from ConfSolve is a floating point type. Floating point variables are in fact part of the MiniZinc language, however they were not implemented in Gecode until March 2013. It would be useful to have floating point variables to model rates or other derivatives of configuration parameters.

A natural addition to ConfSolve is a first-class IP address type and corresponding bitwise operators, namely AND and OR. It is common to apply bitwise masks to IP addresses, for example to identify a subnet to which routing rules apply. MiniZinc does not have any bitwise operators, nor a bit vector type. One solution would be to translate an IP address to an **int** which is an index into an array in which all IP addresses values are allocated. The values themselves would be sets of 1..32 for IPv4 and 1..64 for IPv6. Bitwise AND would then be implemented using MiniZinc's set intersection operator and bitwise OR using set union.

It is possible to translate ConfSolve into other solver input languages such as SMT. The semantics of ConfSolve, defined in terms of MiniZinc, could be generalised to finite-domain CSP with first-order constraints in order to decouple the two languages. A translation from ConfSolve to SMT would depend on the feature set of the solver. Solvers which can handle nested records do not have to perform flattening. SMT is typically more expressive than MiniZinc, providing richer constructs such uninterpreted functions, which could be used to represent object properties.

The type system of ConfSolve is an interesting area for further work. We can see the potential for user-defined primitive (*i.e.*, non-object) types which are a combination of an underlying type and a predicate, for example the type $\{x \in \mathbb{Z}^+ \mid x \mod 256 = 0\}$ would be useful for modelling memory size. This is an example of a *refinement type*. Finally, as is often the case in functional programming languages, it may be convenient to infer types from their values where possible, using Hindley-Milner unification. This would avoid the need to provide types for constants, thus `var x = 1` would result in `x` being declared as an integer.

Bibliography

Ozgur Akgun, Ian Miguel, Chris Jefferson, Alan M Frisch, and Brahim Hnich. Extensible automated constraint modelling. In *AAAI Conference on AI*, 2011.

Kyriakos Anastasakis, Behzad Bordbar, Geri Georg, and Indrakshi Ray. Uml2alloy: A challenging model transformation. *Model Driven Engineering Languages and Systems*, pages 436–450, 2007.

P. Anderson. Programming the Datacentre. *The Rise and Rise of the Declarative Datacentre, Microsoft Technical Report MSR-TR-2008-61*, page 4, 2008.

P. Anderson and A. Scobie. Large scale Linux configuration with LCFG. In *Proceedings of the Atlanta Linux Showcase*, pages 363–372. USENIX, 2000.

Paul Anderson, Patrick Goldsack, and Jim Paterson. SmartFrog meets LCFG: Autonomous reconfiguration with central policy control. In *Proceedings of the 17th conference on Large Installation System Administration Conference*, pages 219–228, 2003.

N. Arshad, D. Heimbigner, and A.L. Wolf. Deployment and dynamic reconfiguration planning for distributed software systems. In *Tools with Artificial Intelligence, 2003. Proceedings. 15th IEEE International Conference on*, pages 39–46, 2003. doi: 10.1109/TAI.2003.1250168.

T. Benoist, B. Estellon, F. Gardi, R. Megel, and K. Nouioua. Localsolver 1. x: a black-box local-search solver for 0-1 programming. *4OR: A Quarterly Journal of Operations Research*, pages 1–18, 2011.

M.E. Brasher and K. Schopmeyer. CIMPLE: An Embeddable CIM Provider Engine. 2006. URL `http://simplewbem.org/whitepaper.pdf`.

M. Burgess et al. CFEngine: a site configuration engine. *USENIX Computing systems*, 8(3):309–402, 1995.

Jordi Cabot, Robert Clarisó, and Daniel Riera. Verification of UML/OCL class diagrams using constraint programming. In *Software Testing Verification and Validation Workshop, 2008. ICSTW'08. IEEE International Conference on*, pages 73–80. IEEE, 2008.

Marco Castaldi, Stefania Costantini, Stefano Gentile, and Arianna Tocchio. A logic-based infrastructure for reconfiguring applications. In Joäo Leite, Andrea Omicini, Leon Sterling, and Paolo Torroni, editors, *Declarative Agent Languages and Technologies*, volume 2990 of *Lecture Notes in Computer Science*, pages 17–36. Springer, 2004.

CFEngine AS. CFEngine 3 tutorial, 2008. URL `http://cfengine.com/manuals/cf3-tutorial/`.

Manuel Clavel, Marina Egea, and Miguel Angel García de Dios. Checking unsatisfiability for ocl constraints. *Electronic Communications of the EASST*, 24, 2010.

Hewlett-Packard Development Company. The smartfrog constraint extensions, 2005. URL `http://www.hpl.hp.com/research/smartfrog/releasedocs/smartfrogdoc/csfExtensions.pdf`.

A. Couch and M. Gilfix. It's elementary, dear Watson: applying logic programming to convergent system management processes. In *Proc. LISA '99*. USENIX, 1999.

N. Damianou, N. Dulay, E. Lupu, and M. Sloman. The Ponder policy specification language. *Policies for Distributed Systems and Networks*, pages 18–38, 2001.

T. Delaet and W. Joosen. PoDIM: A language for high-level configuration management. In *Proceedings of LISA '07*. USENIX, 2007.

Thomas Delaet, Paul Anderson, and Wouter Joosen. Managing real-world system configurations with constraints. In *Networking, 2008. ICN 2008. Seventh International Conference on*, pages 594–601. IEEE, 2008.

Distributed Management Task Force. Cim tutorial, 1999. URL `http://www2.informatik.hu-berlin.de/~xing/Lib/cim-tutorial/extend/spec.html`.

Distributed Management Task Force. Common information model (CIM) standards, 2010a. URL `http://www.dmtf.org/standards/cim/`. Available from `http://www.dmtf.org/standards/cim/`.

Distributed Management Task Force. Web services for management, 2010b. URL `http://www.dmtf.org/standards/wsman/`.

Distributed Management Task Force. Web-based enterprise management (wbem), 2010c. URL `http://www.dmtf.org/standards/wbem`.

Jeffrey Fischer, Rupak Majumdar, and Shahram Esmaeilsabzali. Engage: a deployment management system. In *Proceedings of the 33rd ACM SIGPLAN conference on Programming Language Design and Implementation*, PLDI '12, pages 263–274, New York, NY, USA, 2012. ACM.

A.M. Frisch, M. Grum, C. Jefferson, B.M. Hernández, and I. Miguel. The design of essence: A constraint language for specifying combinatorial problems. In *Proc., Twentieth International Joint Conference on Artificial Intelligence (IJCAI)*, 2007.

Haris Gavranović, Mirsad Buljubašić, and Emir Demirović. Variable neighborhood search for google machine reassignment problem. *Electronic Notes in Discrete Mathematics*, 39:209–216, 2012.

Gecode Team. Gecode: Genetic constraint development environment, 2006. Available from `http://www.gecode.org`.

Alfonso Gerevini and Ivan Serina. Lpg: A planner based on local search for planning graphs with action costs. In *Proc. of the Sixth Int. Conf. on AI Planning and Scheduling*, pages 12–22, 2002.

Carmen Gervet. Interval propagation to reason about sets: Definition and implementation of a practical language. *Constraints*, 1(3):191–244, 1997.

F. Hermenier, S. Demassey, and X. Lorca. Bin repacking scheduling in virtualized datacenters. In *Principles and Practice of Constraint Programming – CP 2011*. Springer, 2011.

Fabien Hermenier, Xavier Lorca, Jean-Marc Menaud, Gilles Muller, and Julia Lawall. Entropy: a consolidation manager for clusters. In *Proceedings of the 2009 ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*, pages 41–50. ACM, 2009.

J.A. Hewson and P. Anderson. Modelling system administration problems with CSPs. In *Proceedings of the 10th International Workshop on Constraint Modelling and Reformulation (ModRef'11)*, pages 73–82, 2011.

J.A. Hewson, P. Anderson, and A.D. Gordon. A declarative approach to automated configuration. In *26th Large Installation System Administration Conference (LISA'12)*, 2012.

Michael Hicks and Scott Nettles. Dynamic software updating. *ACM Trans. Program. Lang. Syst.*, 27(6):1049–1096, November 2005.

T. Hinrich, N. Love, C. Petrie, L. Ramshaw, A. Sahai, and S. Singhal. Using object-oriented constraint satisfaction for automated configuration generation. *Lecture notes in computer science*, pages 159–170, 2004.

D. Jackson. Alloy: a lightweight object modelling notation. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 11(2):256–290, 2002.

D Jackson. Software abstractions: Logic, language and analysis, revised edn, 2012.

Narendra Jussien, Guillaume Rochart, Xavier Lorca, et al. Choco: an open source java constraint programming library. In *CPAIOR'08 Workshop on Open-Source Software for Integer and Contraint Programming (OSSICP'08)*, pages 1–10, 2008.

L. Kanies. ISconf: Theory, practice, and beyond. In *Proceedings of LISA XVII*, pages 115–123, 2003.

Leonidas Lymberopoulos, Emil Lupu, and Morris Sloman. Ponder policy implementation and validation in a cim and differentiated services framework. In *Network Operations and Management Symposium, 2004. NOMS 2004. IEEE/IFIP*, volume 1, pages 31–44. IEEE, 2004.

Kim Marriott, Nicholas Nethercote, Reza Rafeh, Peter J Stuckey, Maria Garcia De La Banda, and Mark Wallace. The design of the Zinc modelling language. *Constraints*, 13(3):229–267, 2008.

Microsoft Corporation. Windows management instrumentation (windows), 2010. URL `http://msdn.microsoft.com/en-us/library/aa394582.aspx`.

S. Mittal and B. Falkenhainer. Dynamic constraint satisfaction problems. In *Proceedings of the Eighth National Conference on Artificial Intelligence*, pages 25–32, 1990.

S. Mittal and F. Frayman. Towards a generic model of configuration tasks. In *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence*, volume 2, pages 1395–1401. IJCAI, 1989.

S. Narain. Network configuration management via model finding. In *Proceedings of the 19th conference on Large Installation System Administration Conference*, page 15. USENIX Association, 2005.

S. Narain, T. Cheng, B. Coan, V. Kaul, K. Parmeswaran, and W. Stephens. Building autonomic systems via configuration. In *Proceedings of IEEE Autonomic Computing Workshop*, 2003.

S. Narain, G. Levin, S. Malik, and V. Kaul. Declarative infrastructure configuration synthesis and debugging. *Journal of Network and Systems Management*, 16(3):235–258, 2008.

Timothy Nelson, Christopher Barratt, Daniel J Dougherty, Kathi Fisler, and Shriram Krishnamurthi. The margrave tool for firewall analysis. In *Proceedings of the 24th Large Installation System Administration Conference*, pages 1–8. USENIX Association, 2010.

N Nethercote. Converting MiniZinc to FlatZinc, 2012. URL `http://www.minizinc.org/downloads/doc-1.5/mzn2fzn.pdf`.

N. Nethercote, P. Stuckey, R. Becket, S. Brand, G. Duck, and G. Tack. MiniZinc: Towards a standard CP modelling language. *Principles and Practice of Constraint Programming (CP 2007)*, pages 529–543, 2007.

Object Management Group. Object constraint language specification 2.0, 2006. URL `http://www.omg.org/spec/OCL/2.0/PDF`.

Puppet Labs. Puppet, 2008. Available from `http://www.puppetlabs.com/puppet/`.

Puppet Labs. Learning — modules and classes, 2011. URL `http://docs.puppetlabs.com/learning/modules1.html`.

L. Ramshaw, A. Sahai, J. Saxe, and S. Singhal. Cauldron: A policy-based design tool. In *7th IEEE International Workshop on Policies for Distributed Systems and Networks (POLICY 2006)*, pages 113–122, 2006.

Mark Richters and Martin Gogolla. On formalizing the uml object constraint language ocl. In *Conceptual Modeling (ER'98)*, pages 449–464. Springer, 1998.

ROADEF. Google ROADEF/EURO challenge 2011-2012: Machine reassignment, 2011. URL `http://challenge.roadef.org/2012/files/problem_definition_v1.pdf`.

F Rossi, P van Beek, and T Walsh. Constraint logic programming. In *Handbook of constraint programming*, chapter 12. Elsevier Science Ltd, 2006.

D. Sabin and E.C. Freuder. Configuration as composite constraint satisfaction. In *Proceedings of the Artificial Intelligence and Manufacturing Research Planning Workshop*, pages 153–161, 1996.

A. Sahai, S. Singhal, R. Joshi, and V. Machiraju. Automated generation of resource configurations through policies. In *IEEE Policy*, 2004a.

A. Sahai, S. Singhal, R. Joshi, and V. Machiraju. Automated policy-based resource construction in utility computing environments. In *the Proceedings of the 2004 IEEE/IFIP Network Operations & Management Symposium (NOMS 2004)*, pages 19–24, 2004b.

I. Shah. A hybrid algorithm for finding minimal unsatisfiable subsets in over-constrained CSPs. *International Journal of Intelligent Systems*, 26(11):1023–1048, 2011.

Parag Singla and Pedro Domingos. Memory-efficient inference in relational domains. In *Proceedings of the 21st national conference on Artificial intelligence*, volume 1 of *AAAI'06*, pages 488–493. AAAI Press, 2006.

Peter J Stuckey, Maria Garcia de la Banda, Michael Maher, Kim Marriott, John Slaney, Zoltan Somogyi, Mark Wallace, and Toby Walsh. The G12 project: Mapping solver independent models to efficient solutions. In *Principles and Practice of Constraint Programming-CP 2005*, pages 13–16. Springer, 2005.

M. Stumptner and A. Haselböck. A generative constraint formalism for configuration problems. *Advances in Artificial Intelligence*, pages 302–313, 1993.

M. Stumptner, G.E. Friedrich, and A. Haselböck. Generative constraint-based configuration of large technical systems. *AI EDAM*, 12(04):307–320, 1998.

E. Torlak and D. Jackson. Kodkod: A relational model finder. *Lecture Notes in Computer Science*, 4424:632, 2007.

William Vambenepe. Microsoft ditches SML, returns to SDM?, 2008. URL `http://stage.vambenepe.com/archives/163`.

W3C Members. Service modeling language, version 1.0, 2007. URL `http://www.w3.org/Submission/2007/SUBM-sml-20070321/`.

Christoph Walther and Stephan Schweitzer. About VeriFun. In Franz Baader, editor, *Automated Deduction - CADE-19*, volume 2741 of *Lecture Notes in Computer Science*, pages 322–327. Springer Berlin / Heidelberg, 2003.

Michel Wermelinger, Antónia Lopes, and José Luiz Fiadeiro. A graph based architectural (re)configuration language. In *Proceedings of the 8th European software engineering conference*, ESEC/FSE-9, pages 21–32, New York, NY, USA, 2001. ACM.

S.R. White, J.E. Hanson, I. Whalley, D.M. Chess, and J.O. Kephart. An architectural approach to autonomic computing. In *Proceedings of the International Conference on Autonomic Computing*, pages 2–9, 2004.

Yang Xu. Modelling cloud infrastructure configuration with answer set programming. Master's thesis, University of Edinburgh, 2012. URL `http://www.inf.ed.ac.uk/publications/thesis/offline/IM121189.pdf`.

Q. Yin, J. Cappos, A. Baumann, and T. Roscoe. Dependable self-hosting distributed systems using constraints. In *Proceedings of the Fourth conference on Hot topics in system dependability*, pages 11–11. USENIX Association, 2008.